Common LISP Modules

VISIT…

LANZAROTE
Caliente.COM

Mark Watson

Common LISP Modules

Artificial Intelligence in the Era of Neural Networks and Chaos Theory

With 35 Illustrations

Springer-Verlag
New York Berlin Heidelberg London
Paris Tokyo Hong Kong Barcelona

Mark Watson
Science Applications International Corporation
10260 Campus Point Drive
San Diego, CA 92121
U.S.A.

Printed on acid-free paper.

Copy produced using the author's PostScript file.

9 8 7 6 5 4 3 2 1

ISBN-13: 978-0-387-97614-3 e-ISBN-13: 978-1-4612-3186-8
DOI: 10.1007/978-1-4612-3186-8

Preface

While creativity plays an important role in the advancement of computer science, great ideas are built on a foundation of practical experience and knowledge. This book presents programming techniques which will be useful in both AI projects and more conventional software engineering endeavors. My primary goal is to entertain, to introduce new technologies and to provide reusable software modules for the computer programmer who enjoys using programs as models for solutions to hard and interesting problems. If this book succeeds in entertaining, then it will certainly also educate.

I selected the example application areas covered here for their difficulty and have provided both program examples for specific applications and (I hope) the methodology and spirit required to master problems for which there is no obvious solution.

I developed the example programs on a Macintosh™ using the Macintosh Common LISP™ development system capturing screen images while the example programs were executing. To ensure portability to all Common LISP environments, I have provided a portable graphics library in Chapter 2.

All programs in this book are copyrighted by Mark Watson. They can be freely used in any free or commercial software systems if the following notice appears in the fine print of the program's documentation: "This program contains software written by Mark Watson." No royalties are required.

The program miniatures contained in this book may not be distributed by posting in source code form on public information networks, or in printed form without my written permission.

I will provide Floppy disks (Macintosh™ or IBM PC™ format) containing the source code to all example programs listed in this book for $12 (order from my address below). This software contained in this book is "as is" with no warranties or proof of correctness; the user of this software accepts all liabilities for its use. I also welcome other correspondence (preferably via electronic mail). I can be contacted on the following public information services:

Compuserve address: 75765,556

BIX: wmark

Home address: 535 Mar Vista Dr. Solana Beach, CA 92075

Acknowledgments

I would like to thank my wife Carol for encouragement while preparing this book; my company, SAIC, for supporting my research in AI and Neural Network technologies; Tim Kraft, George Works, and David Rumelhart for many discussions concerning neural network technology; Bob Beyster, Carl Rindfleisch, John Benepe, John Thompson, Bobby Hunt, John Penhune, Tom Bache, Duane Knize, Larry Kull, Larry Hunt, and Jim Martin for supporting my AI research at SAIC; Carol Watson, Cris Kobryn, and Tim Kraft for reviewing rough drafts of this book; my father, Ken Watson, for the use of his laser printer in preparing this book; my agent, Bill Gladstone, and his assistant, Matt Wagner; my editors Gerhard Rossbach, Suzanne Anthony, and Kenneth Dreyhaupt; and Nancy Wilson for the excellent job editing my book. I would also like to thank the authors of all the books in the bibliography section. I have learned a great deal from them.

Contents

1
Introduction

Computer programmers have tried to simulate intelligent behavior with programs since the invention of computers. It is easy to be sympathetic with pioneering computer scientists who predicted human-class cognitive performance out of computers when their first experiences showed that even early computers' arithmetic ability far surpassed that of humans. With the benefit of hindsight, it is clear that artificial intelligence (AI) practitioners promised too much too soon.

We now know that human beings far outclass even supercomputers in pattern recognition and associative memory recall. A common (and much enjoyed) debate deals with the likelihood of AI hardware/software systems rivaling humans in cognitive tasks. This debate will not be covered in this book! The reader of this *book* will gain insight into the creative process needed to break down old technology barriers and will develop an optimistic "can do" attitude.

1.1 Overview

Except where noted, the chapters in this book can be referenced independently. Chapter 2 contains the device independent graphics primitives used in this book. The neural network simulator developed in chapter 3 forms a conceptual basis for all subsequent treatment of neural networks in chapters 4 through 7. The material in chapter 9 is based on the natural language parsing techniques presented in chapter 8.

Each chapter is generally structured as: background information, theory, a discussion of sample programs, program listings, program output, additional information on the execution of the sample programs, and suggested projects. There are five sections in this book:

1.1.1 Neural networks

Chapter 3 provides background to the theory and usefulness of neural networks, and presents a neural network simulator which will be used in a speech recognition system in chapter 5 and a handwriting recognition system in Chapter 6. Chapter 4 introduces associative memory systems using Hopfield neural networks. Chapter 7 introduces the biologically inspired neural network model ART2 of Stephen Grossberg and Gail Carpenter.

1.1.2 Natural language processing

Chapter 8 introduces syntax-based parsing techniques with an animated parsing program. Chapter 9 adds semantic processing to syntactic analysis and compares and contrasts the example library database query program to programs using Conceptual Dependency theory.

1.1.3 Expert-systems

Expert-systems, usually embedded in larger software systems, are becoming a standard software engineering tool. Sample programs in chapter 10 demonstrate forward and backward chaining techniques. Advice is offered in the evaluation of commercial expert-system shells.

1.1.4 Search

A software module for performing A* heuristic network searches is developed in chapter 11. A simple but effective chess playing program is developed in chapter 12.

1.1.5 Chaos theory

Chapter 13 introduces chaos theory. Chapter 14 contains sample programs for generating Mandelbrot set plots and iterated function system plots.

1.2 Prerequisites

Ideally this book should be read with the benefit of a computer system with a LISP compiler since all chapters contain, as program miniatures, small programs written to illustrate an idea or concept. Note: All programs are implemented in Common LISP. However, the examples should be easily portable to other LISP dialects. The neural network, and fractal examples can be ported to other languages like C and Pascal since they do not rely on the symbolic features of the LISP programming language.

This book assumes a reading knowledge of LISP and the implementation of a set of graphics primitives described in chapter 2 for creating a graphics window and performing simple graphics operations in this window. This set of primitives can be written in about two pages of code in any Common LISP system that provides simple graphics support; once this library is written, all of the miniatures presented in this book should execute in all Common LISP environments using a LISP listener window for user interaction with graphics going to a separate window.

1.3 Conventions

Key words appear in **bold** when first introduced. All program literals will be in *italic* when they appear in text and in comments in the program listings. All program test outputs have user responses in **bold** text with program output shown in normal typeface. Long program comments will be blocked off separately and indented to the same column as the surrounding text. Short comments appear on the same line as LISP code. Example programs are printed in Helvetica type.

2
Basic Software Tools: Machine-Independent Graphics

One of the best uses for a high-level language like Common LISP is in the rapid prototyping of window-based user interfaces, now an accepted part of the software development process. Common LISP is especially appropriate for developing user interfaces when augmented with one of the current object oriented programming (OOP) packages: Object Lisp, Common LISP object system (CLOS), or Portable Common Loops (PCL).

However, one requirement for a programming language book is to enable readers to effortlessly get the sample programs in a book up and running (with no modifications!) on their computer and within their language environment. For this reason, the example programs in this book will use a simple set of graphic primitives and will not use any OOP language extensions despite their obvious benefit. Once the graphics library described below is implemented, all example programs should execute without modification on any system.

The examples in this book use the following graphics primitives:

init-plot	– Creates a graphics window
plot-fill-rect	– Fills a rectangle with a gray-scale value
plot-size-rect	– Plots a rectangle whose size is specified by a gray-scale value
clear-plot	– Clears the graphics window
pen-width	– Sets the default line-plotting width
plot-frame-rect	– Draws a rectangle
plot-line	– Draws a line
show-plot	– Makes visible and unobscured the graphics window
plot-string	– Plots a string of text
plot-string-bold	– Plots a string of bold text
plot-string-italic	– Plots a string of italic text
plot-mouse-down	– If the mouse is clicked, returns the x-y click position

Here is an example implementation of these primitives for the Macintosh running Macintosh Common LISP:

```
;;
; Common plot routines for Common LISP Compatibility — for MACINTOSH
;
; Externally callable functions:
;
; init-plot( )                             ;; creates a graphics window
; plot-fill-rect(x y xsize ysize value)    ;; fills a rectangle with a gray-scale value
; plot-size-rect(x y xsize ysize value)    ;; plots a rectangle value pixels wide
; clear-plot( )                            ;; clears the graphics window
; pen-width (nibs)                         ;; sets the pen drawing width
; plot-frame-rect(x y xsize ysize)         ;; plots a frame rectangle
; plot-line(x1 y1 x2 y2)                   ;; plots a line between two points
; show-plot( )                             ;; shows graphics window
; plot-string(x y str)                     ;; plots a string at position (x y)
; plot-string-bold(x y str)                ;; plots a bold string at position (x y)
; plot-string-italic(x y str)              ;; plots a italic string at position (x y)
; plot-mouse-down( )                       ;; returns position of mouse click
;;

;;
; Initializes a standard plot window:
;;

(defun init-plot (&optional (title "Plot Window") (xSize 250) (ySize 250) )
  (setq *w*
     (oneof *window* :window-title title
                     :window-size (make-point xSize ySize)
                     :window-type :document-with-zoom)))

;;
; Fills in a rectangle with one of five gray-scale values:
;;

(defun plot-fill-rect (x y xsize ysize pattern)
  (setq pattern (truncate pattern))
  (let ((ppp *black-pattern*))
    (if (< pattern 1)
      (setq ppp *white-pattern*)
      (if (equal pattern 1)
        (setq ppp *light-gray-pattern*)
        (if (equal pattern 2)
          (setq ppp *gray-pattern*)
          (if (equal pattern 3)
            (setq ppp *dark-gray-pattern*)
            (setq ppp *black-pattern*)))))
```

```
   (ask *w* (fill-rect ppp x y (+ x xsize) (+ y ysize)))))

;;
; Makes a black rectangle of size proportional to val. This is an alternative
; to using function plot-fill-rect for showing graphically the value of a number.
;;

(defun plot-size-rect (x y xsize ysize val)
  (setq val (min val xsize))
  (ask *w* (erase-rect x y (+ x xsize) (+ y ysize)))
  (ask *w* (fill-rect *black-pattern* x y (+ x val) (+ y val))))

;;
; Clears (erases) the plot window:
;;

(defun clear-plot ()
  (ask *w* (fill-rect *white-pattern* 0 0 400 400)))

;;
; Sets the drawing size for the pen:
;;

(defun pen-width (nibs)
  (ask *w* (set-pen-size (make-point nibs nibs))))

;;
; Frames a rectangle of size (xsize ysize) at position (x y):
;;

(defun plot-frame-rect (x y xsize ysize)
  (let ((x2 (+ x xsize))
        (y2 (+ y ysize)))
    (ask *w* (frame-rect x y x2 y2))))

;;
; Draws a line between two points:
;;

(defun plot-line (x1 y1 x2 y2)
  (ask *w* (move-to x1 y1))
  (ask *w* (line-to x2 y2)))
```

```
;;
; Shows plot window if it is obscured:
;;

(defun show-plot ()
  (ask *w* (window-select)))

;;
; Plots a string at position (x y):
;;

(defun plot-string (x y str &optional (size 10))
  (setq x (truncate x)  y (truncate y))
  (ask *w* (set-window-font (list "Times" :plain size)))
  (ask *w* (move-to x y))
  (princ str *w*))

;;
; Plots a string in bold font at position (x y):
;;

(defun plot-string-bold (x y str &optional (size 12))
  (setq x (truncate x)  y (truncate y))
  (ask *w* (set-window-font (list "Times" :plain :bold size)))
  (ask *w* (move-to x y))
  (princ str *w*))

;;
; Plots a string in italic font at position (x y):
;;

(defun plot-string-italic (x y str)
  (setq x (truncate x)  y (truncate y))
  (ask *w* (set-window-font '("Times" :plain :italic 12)))
  (ask *w* (move-to x y))
  (princ str *w*))

;;
; Tests for a mouse down event (returns nil if the mouse button is
; not being held down when this function is called;  returns a list
; of the x and y screen coordinates relative to the plot window
; origin if the mouse button is being held down while this
; function is being called):
;;
```

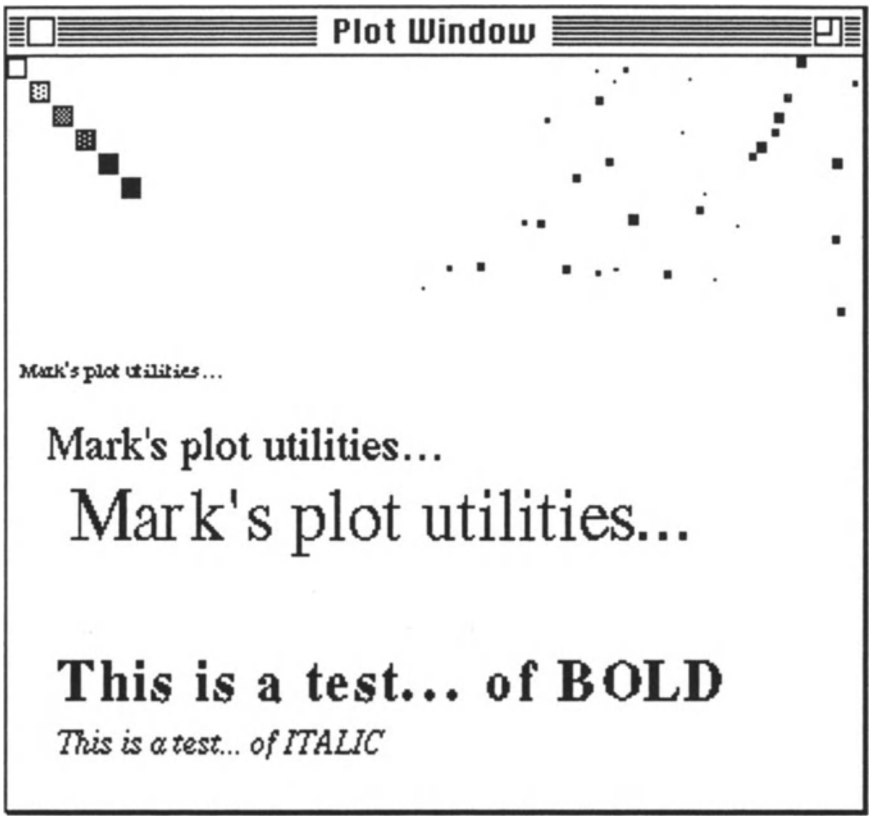
```
(defun plot-mouse-down ()
 (mouse-down-p))

;;
; A simple test program:
;;

(defun test ()
 (show-plot)
 (clear-plot)
 (dotimes (i 6)
  (plot-fill-rect
   (* i 9)
   (* i 9)
   8 8
   i)
  (plot-frame-rect (* i 9) (* i 9) 8 8))
 (dotimes (i 50)
  (plot-size-rect
   (+ 160 (random 200)) (random 100) (random 20) (random 20) (random 5)))
 (dotimes (i 4)
  (plot-string (* i 10) (+ 150 (* i 22)) "Mark's plot utilities..."))
 (plot-string-bold 20 260 "This is a test... of BOLD")
 (plot-string-italic 20 280 "This is a test... of ITALIC"))
```

Before you implement these primitives within your Common LISP environment, note the following:

1. If you are not using Macintosh Common LISP, it will be necessary to recode each of the above functions using graphics functions provided by your Common LISP vendor. You can start by implementing *init-plot*, *plot-line*, and *plot-string*. The remaining functions can be "stubbed" and implemented later as you need them.

2. If the graphics library provided with your Common LISP environment does not support different fonts, use the function *plot-string* for drawing characters on the screen.

The screen image in Figure 2.1 shows the results of executing the function *test* at the end of the preceeding listing.



Mark's plot utilities...

Mark's plot utilities...

Mark's plot utilities...

This is a test... of BOLD

This is a test... of ITALIC

Figure 2.1. Sample screen image showing the execution of function *test* in the graphics primitives plot library.

3
The Substrates of Intelligence, a Neural Network Primer

What is an artificial neural network? How do artificial neural networks compare with conventional computers and traditional massively parallel computers, and when are they more useful? What are possible applications of neural network technology? These questions will be answered in the beginning of this chapter, followed by the presentation of an engineering model based on equations which characterize the behavior of one popular class of neural networks for **supervised** learning. Supervised learning uses both input training patterns and desired system output patterns for neural network training. This chapter then provides a simple program demonstrating how to write and run artificial neural network simulators followed by a listing of a complete production-capable neural network simulator. Examples show how to set up training data for and run this complete simulator (which is used for speech recognition in chapter 5 and reading handwritten characters in chapter 6). This chapter ends with more suggested projects and hints for their solution.

3.1 Background

Neural networks are systems of very simple processing elements that are massively interconnected. Long term memory or information content in neural networks is typically stored in the state of the interconnections, not the processing elements themselves. Currently artificial neural systems are computer simulations of systems containing large numbers of massively interconnected processors. Neural networks are not programmed in the traditional sense; they are **trained** by exposure to data values specifying desired system outputs for given system inputs.

Many AI problems can be solved in principle with known algorithms, but not in practice because of the limited processing capabilities of the Von Neuman computer model (as implemented in single processor IBM PCs, Apple Macintoshs, DEC VAXs, CRAYs, etc.). These conventional computer systems are best used for applications which are procedurally oriented like business and nonparallel scientific calculations. By contrast, neural systems exploit the inherent parallelism in many pattern-matching and recognition tasks. In the future they will be used to augment conventional computers, greatly speeding up pattern matching and cognitive tasks.

Neural systems are very fault tolerant; they can be partially destroyed and still function with some degradation of performance. This fault tolerance will someday make it possible to construct neural computers with millions of neurons with tens of millions of interconnections using three-dimensional integrated circuits. In addition to increased processing speed, an advantage of building a system using neural network chips over conventional integrated circuits is the tolerance these chips have for fabrication defects. If proposed neural network chips had to have perfectly functioning components (like the digital VLSI chips in your microcomputer), it would be impossible to manufacture neural chips with enough components to solve interesting problems with sufficient chip yield.

3.1.1 Existing Parallel Computers

The idea of using a large number of processors working in parallel is not new. Currently, under DARPA sponsorship, several interesting parallel machines are being constructed. These machines range from modest parallelism (e.g., the BBN Butterfly machine with 16, 64, or 256 processors) to massive parallelism (e.g., the Thinking Machines Inc. Connection machine with 64,000 to 1,000,000 individual processors). The construction of real neural computers in the future will be another alternative approach to parallelism.

Every computer system which uses a large number of individual processors must be able to quickly transfer information between those processors. It is this transfer of data between processors which prevents *N* processors from solving most application problems *N* times faster than a single processor. Several clever topologies have been devised for connecting many Von Neuman-type processors together. The BBN Butterfly machine connects its processors in a "butterfly" pattern similar to the order of calculations in a radix four Fast Fourier Transform (FFT). Figure 3.1 shows a Butterfly topology in which processing elements with local memory appear as rectangles and communication links appear as straight lines; the processors are laid out so that those on the right edge of the diagram have connections back to the processors on the left edge of the diagram, as if they were on the curved surface of a cylinder.

The maximum communication time between any two processing nodes in a Butterfly computer is proportional to log(*N*), where *N* is the total number of processors in the system. The Butterfly computer has the rather nice property of worst-case communication time between processors degrading slowly as more processors are added to the system. Its disadvantage is that whenever you add processors to a system, you must do so by multiplying the number of processors by 4.

The Connection Machine computer handles the interprocessor communication problem by organizing the processing cells around a grid of communication centers connected via a packet switched network. Each communication center services 16 processing cells. The processors in the Connection Machine are configured in software to match the data structures used in calculations. Two examples of logically reconfiguring the Connection Machine to match data structure requirements are simulation of cellular automata (processors logically configured to lie in a two-dimensional grid with direct communication between adjacent processors) and the configuration of the processors into a binary tree for algorithms that calculate values

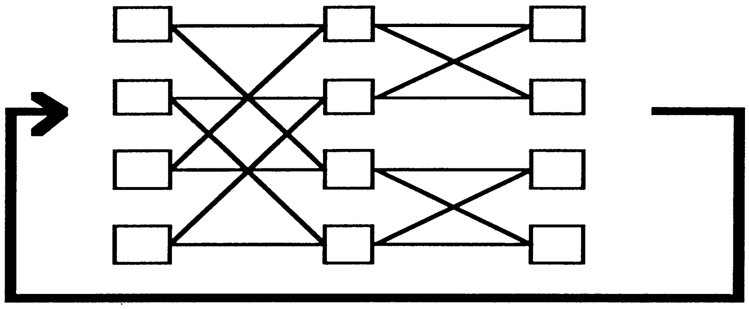


Figure 3.1. A Partial Butterfly Topology: processors on the right side of the figure are connected back to the column of processors on the left side of the figure

at tree leaves and that back up the results of these calculations to the root of the binary tree.

Both the Butterfly and Connection Machines have ingenious software support systems to help software developers handle the complexity of data communication between processors. However, it is often difficult to write software to take full advantage of highly parallel computers. Neural networks offer a technology for dealing with large numbers of parallel processing elements without specialized software requirements.

3.1.2 Neural Network Applications

The training of a neural network to recognize spoken words as in the chapter 5 example program is an example of a good neural network application, that is, one that takes advantage of the characteristics of this technology. The spectral features from speech data can be extracted for use as the input to a neural network while the training output pattern is a specification of the word currently being spoken. Once spectral features are extracted from speech, phoneme or word recognition is an example of the type of fine-grained data parallelism neural networks are good at.

Another example of fine-grain data parallelism in a pattern-matching application is the recognition of handwritten characters. Preprocessing is required to isolate and center individual characters, which neural networks can then recognize. A complete handwriting recognition system for recognizing the numbers from 0 to 9 appears in chapter 6. This system can be extended to recognize additional characters like the letters of the alphabet.

Both of these application examples show the importance of using neural network technology where it is appropriate to do so, and to combine neural networks with conventional procedural/algorithmic processing to build larger systems. Neural networks can be best used as components of large systems to handle pattern matching tasks. It is important to avoid the use of neural network technology in cases like algorythmic processing where conventional software solutions are easier to implement and offer better performance.

3.2 Model for Supervised Learning

To begin our discussion of a model for supervised learning neural networks, we will define several terms. A **neuron** is a simple processor which sums the output from one or more other neurons, applies some form of transfer function to this sum, and outputs this transformed sum to the inputs of other neurons. The output of a neuron is called its **activation energy**. A **connection weight** is a numeric quantity which determines how much of a neuron's output value reaches the input to the next neuron to which it is connected. A connection weight is defined by a source neuron, a target neuron, and a numeric weighting factor. In our model, we use real or floating point (noninteger) numbers to characterize activation energies and connection weights. For example, a connection weight of 0.5 allows more activation energy to flow between the output of one neuron and the input of another than a connection weight of 0.012. Connection weights are often referred to simply as **weights** or **connections**. A **layer** of neurons is a set of logically grouped neurons. Layers of neurons are frequently referred to as **slabs**. Our neural models developed in this chapter will always have at least one input layer and one output layer; often, they will have one or more **hidden** layers, which are defined as having no direct connections to neural network input or output signals.

Our model of a neural network has input neurons, which accept input information (or data), and output neurons, which provide external output information. We view a neuron as having an activation energy in the range (0.0, 1.0). One way to visualize what our neural model is doing is to imagine that we have N input neurons which form a space of N dimensions. We are in effect building a nearest-neighbor classifier in N dimensions.

Imagine that before we start giving the system any training examples each point in this N-dimensional space has the same constant value. As the system self-adapts during training, we form minima in our N-dimensional space centered around points whose coordinates correspond to input data sets (each sample input vector containing N values defines a single point in this N-dimensional space). These minima are often called **basins of attraction**. When we give the system a new set of inputs, we are in effect specifying a point in this N-dimensional space which lies on a curved hyper-surface; we "roll down hill" to a minima formed by the training data set which (we hope) is the closest match to this new input. Figure 3.2 shows this process in one dimension, a neural network with only one input neuron, which can have an output activation energy in the range (0.0, 1.0) before training. Three basins of attraction are created during the training process from three training data values.

After training, any input neuron value in zone 1 will "fall into" minima 1, any value in zone 2 will fall into minima 2, and any input value in zone 3 will fall into minima 3. For example, zone 2 corresponds to an input activation value in the range (0.25, 0.4); when the activation energy of the single input neuron is set to 0.3, it is categorized as being in zone 2. If we have 100 input neurons, the process is exactly the same, only we have 100 directions in a 100-dimensional space.

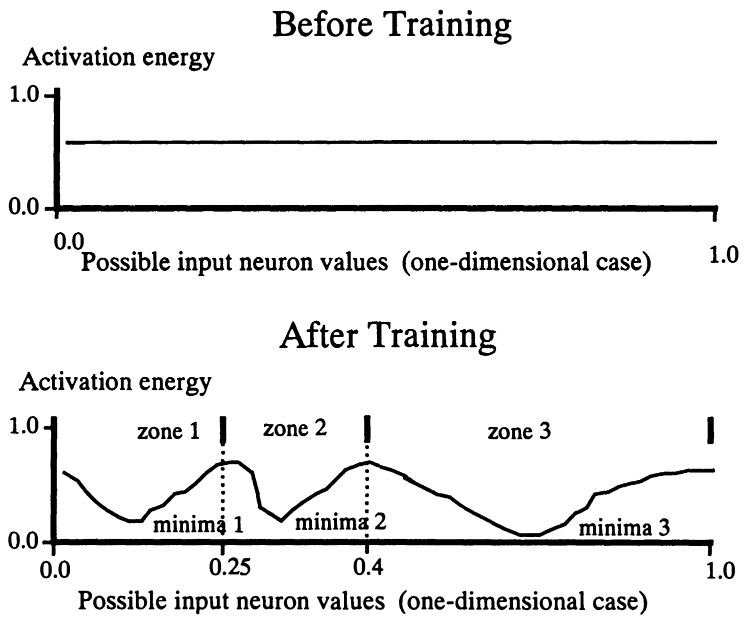


Figure 3.2. Nearest Neighbor Match (One-dimensional case)

3.3 Simple Neural Network

To quickly get an idea of how feed-forward and backwards error propagation (or Delta Rule) networks learn to recognize patterns, we will look at a simple two-layer (one input and one output layer, with no hidden layers) network in which each output neuron is directly connected to each input neuron by a connection characterized by a weight, as seen in Figure 3.3. The larger the value of a weight, the stronger the connection is between the two connected neurons. The following equation gives a value commonly referred to as the **sum of products** for output neuron number *output*:

$$SumProducts_{output} = \sum_{input=0}^{sizeInput} Input_{input} * Weight_{input,output}$$

Here, $Weight_{input,\ output}$ is the numerical value for the connection strength between input neuron *input* and output neuron *output*. The output activation energy is obtained by applying the Sigmoid squashing function to the sum of products:

$$Output_{output} = Sigmoid\ (SumProducts_{output})$$

The shape in Figure 3.4 of the *Sigmoid* curve shows why this is referred to as a squashing function. The Sigmoid function can be implemented as follows:

```
(defun Sigmoid (x)
  (/ 1.0 (+ 1.0 (exp (-x)))))
```

Delta rule neural networks initially have all weights set to small random numbers. The networks are trained by setting the activation energy of each input neuron to a specified training value. Each input neuron passes its activation energy through a weight to an output neuron as seen in Figure 3.3. The weights in a network are *slowly* modified in response to training data. The learning rules for modifying the weights in a network in response to training data are quite straight forward; for every output neuron (in the output slab), calculate its error signal (a small error signal at an output neuron indicates that the output activation energy closely matches the target output activation value):

$$Error_{output} = (target_{output} - Output_{output}) * Sigmoid^{P}\ (SumProducts_{output})$$

The target output for each output neuron is specified in the training data with the corresponding input neuron activation values. *Sigmoid^P* is the derivative of the squashing *Sigmoid* function as seen in Figure 3.5. The derivative of the Sigmoid function can be implemented as follows:

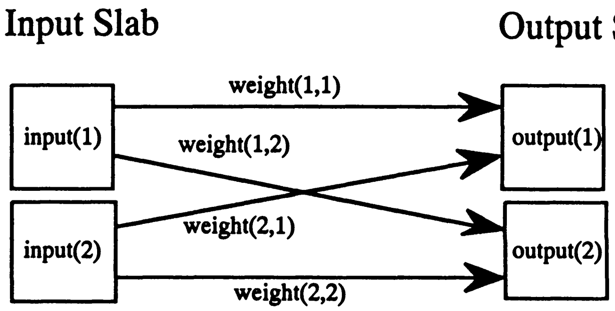


Figure 3.3. Two Layer Delta Rule Network: weight(i,j) connects input neuron with index i to output neuron with index j

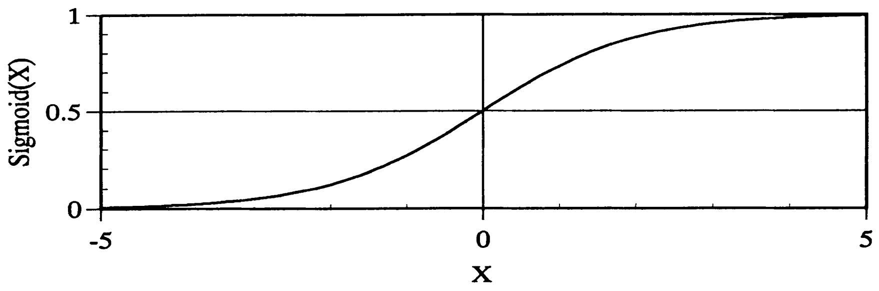


Figure 3.4. Sigmoid Activation Function.

```
(defun dSigmoid (x)
  (let ((temp (Sigmoid x)))
    (* temp (- 1.0 temp))))
```

The shape of the derivative of the squashing function reduces the effective calculated error when the sum of products term for a neuron is very small or very large. Given this error signal, the delta weight change between input neuron *input* and output neuron *output* is simply:

$$DeltaWeight_{input,\,output} = LearningRate * Error_{output} * Ouput_{output}$$

3.3.1 Implementation of the 2-Layer Delta Rule Model

The short example program in this section is an implementation of the equations presented in section 3.3 for activation energy flow through a delta rule neural network and for calculating the weight modifications needed to *learn* the desired training patterns.

The function *makeDelta2* has two input arguments: the size (in neurons) of the input layer and the size of the output layer. It returns as its value a list containing arrays of: network size, input activation values, output activation values, connection strength weights, storage for calculating the sum of products for each output neuron,

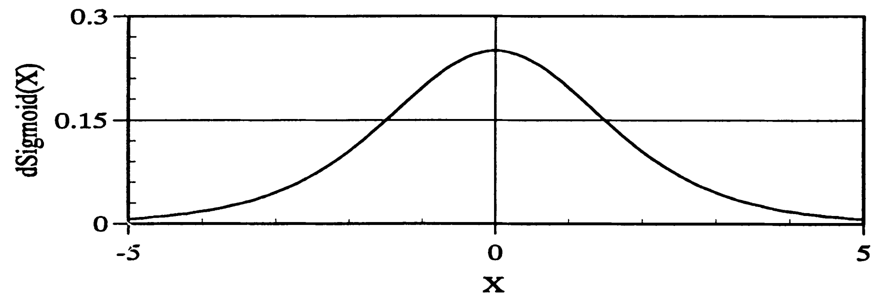


Figure 3.5. Derivative of the Sigmoid Function

storage for calculating the error signal at each output neuron, and storage to sum up the calculated weight changes over all training data sets.

The function *Train2* takes two arguments: a network description list (as returned from *makeDelta2*) and a list of input/ouput training data values. Function *Train2* returns a value for the average error signal at each output neuron; call function *Train2* repetitively until this error signal gets small. The example code fragment at the end of this program listing calls function *Train2* 51 times.

The function *Recall2* takes two arguments: a network description list (as returned from *makeDelta2*) and a list of input neuron values. This function propagates these input values from the input layer neurons to the output layer neurons using the equation for the sum of products and the sigmoid function.

Study the preceeding equations for delta rule learning in neural networks and the short example program in this section. Later a complete simulator for delta rule neural networks will be presented that is more suitable for general experimentation and production use. As you will see, using this general purpose simulation package is fairly straightforward, but the program is complex and will be easier to understand once you study the simple program listed below.

```
;;
;      Simple Delta Rule Neural Network Simulator.
;      This model supports only input and out layers,
;      with no hidden layers as seen in figure 3.3.
;
;      Copyright 1990 by Mark Watson.
;;

;;
; In order to make this example simulator modular and
; reusable, we encapsulate all data for a 2-layer
; delta rule neural network in one LISP data value.
; The following function creates a new network given
; the desired number of input and output neurons.
;;

(proclaim '(special *learning-rate* temp))

(defun makeDelta2 (numInputs numOutputs &aux theNetwork)
 (setq
  theNetwork
  (list
   (list numInputs numOutputs)
   (make-array numInputs :element-type 'float)  ; Input[numInputs]
   (make-array numOutputs :element-type 'float) ; Output[numOutputs]
   (make-array (list numInputs numOutputs)  ; Weight[numInputs][numOutputs]
          :element-type 'float)
   (make-array numOutputs :element-type 'float) ; SumProduct[numOutputs]
   (make-array numOutputs :element-type 'float) ; Error[numOutputs]
   (make-array (list numInputs numOutputs)
          :element-type 'float))) ; DeltaWeight[numInputs][numOutputs]
 ;;
 ; Initialize the network connection strength weight values to random values:
 ;;
 (dotimes (in numInputs)
   (dotimes (out numOutputs)
     (setf (aref (Weight theNetwork) in out) (- (* (random 1000) 0.001) 0.5))))
 theNetwork)

;;
; Functions to return network components from the list structure
; defining our 2-layer delta rule networks:
;;
(defun NetWorkSize (theNetwork) (car theNetwork))
(defun Input (theNetwork) (cadr theNetwork))
(defun Weight (theNetwork) (cadddr theNetwork))
(defun Output (theNetwork) (caddr theNetwork))
(defun SumProduct (theNetwork) (cadddr (cdr theNetwork)))
(defun aError (theNetwork) (cadr (cddddr theNetwork)))
(defun DeltaWeight (theNetwork) (caddr (cddddr theNetwork)))

(setq *Learning-Rate* 0.6) ;; rate of connection-strength modification

;;
; Add a small amout of random noise to a list of numbers:
;;

(defun addNoiseToListOfNumbers (l)
 (defun tweak (x)
   (+
    x
    (- (* (random 1000) 0.0001) 0.05)))
 (mapcar #'tweak l))

;;
; Perform training on our neural network data object using
; the list of lists of training values supplied in the second argument:
;;

(defun Train2 (theNetwork trainingValues
        &aux (numInputs (car (NetWorkSize theNetwork)))
           (numOutputs (cadr (NetWorkSize theNetwork)))
```

```
                (Inputs (Input theNetwork))
                (Outputs (Output theNetwork))
                (Weights (Weight theNetwork))
                (SumProducts (SumProduct theNetwork))
                (Errors (aError theNetwork))
                (DeltaWeights (DeltaWeight theNetwork))
                (RMSerror 0.0))
 ;;
 ; Start out by zeroing the DeltaWeight array. DeltaWeight will be
 ; used to accumulate the delta weight changes for each connection
 ; strength weight over all supplied training values;  the Weight
 ; array is updated after one pass through all supplied training
 ; values (this is the optimum methodology. Modifying the weights
 ; after each individual training input/output example would require
 ; far more training time).
 ;;
 (dotimes (in numInputs)
  (dotimes (out numOutputs)
   (setf (aref DeltaWeights in out) 0.0)))
 ;;
 ; Loop over each input/output training example:
 ;;
 (dolist (inputOutputPair trainingValues)
  (let ((inputList (car inputOutputPair))
       (targetOutput (cadr inputOutputPair))
       tempError)

    ;;
    ; Add some random noise to the training inputs:
    ;;
   (setq inputList (addNoiseToListOfNumbers inputList))

    ;;
    ;  Recall2 propagates these input values
    ; through the network and calculate the output activation values:
    ;;
   (Recall2 theNetwork inputList)

    ;;
    ; Calculate an error for each output neuron scaled
    ; (using the derivative of the Sigmoid function)
    ; by the magnitude of the Sumproducts:
    ;;
   (dotimes (out numOutputs)
    (setq tempError  (- (nth out targetOutput) (aref Outputs out)))
```

```
        (setf
         (aref Errors out)
         (*
          tempError
          (SigmoidP (aref SumProducts out))))
        (setq RMSerror (+ RMSerror (* tempError tempError))))

      ;;
       ; Save the delta connection-strength contribution due
       ; to this input/output training pair:
       ;;
      (dotimes (in numInputs)
        (dotimes (out numOutputs)
          (setf
           (aref DeltaWeights in out)
           (+
            (aref DeltaWeights in out)
            (*
             *Learning-Rate*
             (aref Errors out)
             (aref Inputs in))))))))

  ;;
   ; After processing all input/output training examples, modify
   ; the connection strength weights:
   ;;
  (dotimes (in numInputs)
    (dotimes (out numOutputs)
      (setf
       (aref Weights in out)
       (+
        (aref Weights in out)
        (aref DeltaWeights in out)))))
  ;;
   ; Return the current RMS error for this training set:
   ;;
  RMSerror)

;;
; Recall2 propagates an input pattern through the network
; and returns an array of output layer neuron activation values
;;

(defun Recall2 (theNetwork inputList
               &aux (numInputs (car (NetWorkSize theNetwork)))
```

```
                (numOutputs (cadr (NetWorkSize theNetwork)))
                (Inputs (Input theNetwork))
                (Outputs (Output theNetwork))
                (Weights (Weight theNetwork))
                (SumProducts (SumProduct theNetwork)))
 ;;
 ; Copy the input values to the input neurons:
 ;;
 (dotimes (in numInputs)
  (setf (aref Inputs in) (nth in inputList)))
 ;;
 ; Calculate the SumProducts array for the output neurons:
 ;;
 (dotimes (out numOutputs)
  (setf (aref SumProducts out) 0.0) ;; zero accumulator
  (dotimes (in numInputs)
   (setf
    (aref SumProducts out)
    (+
     (aref SumProducts out)
     (*
      (aref Inputs in)
      (aref Weights in out))))))

 ;;
 ; Use the Sigmoid function to convert the SumProducts array to
 ; the array Outputs (output activation energy):
 ;;
 (dotimes (out numOutputs)
  (setf
   (aref Outputs out)
   (Sigmoid (aref SumProducts out))))
 Outputs)

;;
; Sigmoid and derivative of Sigmoid functions:
;;

(defun Sigmoid (x)
 (/ 1.0 (+ 1.0 (exp (- x)))))

(defun SigmoidP (x)
 (let ((temp (Sigmoid x)))
  (* temp (- 1.0 temp))))
```

```
;;
;THROW AWAY TEST CODE:
; (specify 2 input and 2 output neurons as seen in figure 3.3)
;;
(setq temp (makeDelta2 2 2))

;;
; Train the network and print total output error every 10 trining steps:
;;
(let (rms)
  (dotimes (n 51)
    (setq rms
        (Train2 temp
              '(((1 0) (0 1)) ((0 1 0) (0 1)))))
    (if (equal (mod n 10) 0)
      (progn
        (princ "Current RMS error: ") ;; This error signnal should decrease
        (princ rms)   (terpri)))))        ;; during training.
```

Once you load this short example program to obtain a network like the one shown in Figure 3.3, you will see the following output:

```
((2 2) #(0.0 0.0) #(0.0 0.0)
 #2a((-7.2000000000000008e-2
    0.26800000000000002)
   (0.36699999999999999
    -0.14700000000000002))
 #(0.0 0.0) #(0.0 0.0)
 #2a((0.0 0.0) (0.0 0.0)))
Current RMS error: 1.0610914686801591
Current RMS error: 0.5182104730027558
Current RMS error: 0.27674694940559991
Current RMS error: 0.19533180994390642
Current RMS error: 0.15563273843058725
Current RMS error: 0.1165165556781852
nil
```

3.3.2 Limitations of the 2-Layer Delta Rule Model

The preceeding program listing presented an implementation of a 2-layer neural network simulator, based on the learning equations presented in section 3.3, that consists of two layer networks as seen in Figure 3.3; each layer (input and output) can have any number of neurons.

The addition of hidden slabs (layers) in a network allows it to learn to recognize more complex types of patterns. Figure 3.6 shows the typical capabilities of delta

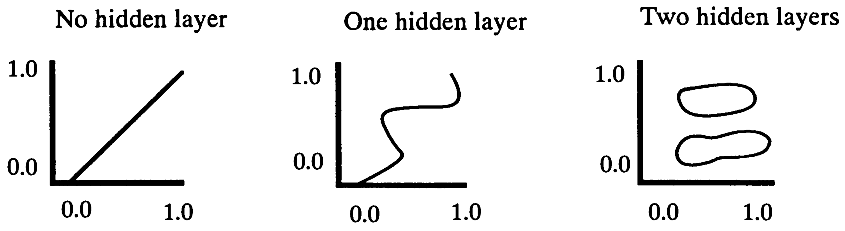


Figure 3.6. Pattern Matching Capability of Two Input Neural Networks with 0, 1, and 2 Hidden Layers. These two dimensional plots represent the range of input values that can be expressed by two input neurons with a range of activation energy values of (0.0, 1.0)

rule networks with zero, one and two layers of hidden neurons. A network with no hidden slabs forms a hyperplane decision-making surface. For example, a system with two inputs can be represented by a two-dimensional plane; the decision surface consists of a one dimensional decision surface (one dimension less than the number of input neurons), or in this case a straight line: these networks can categorize inputs into two sets of points, each set being on one side of the straight line. One hidden slab allows a curved decision surface, but the surface is continuous (unbroken): these networks can separate points into 2 sets separated by an arbitrarily complex unbroken curved line. Two or more hidden slabs allow arbitrary decision surfaces with separate *islands*: these networks can create arbitrary decision surfaces and implement mappings of arbitrary complexity.

In general, networks with one hidden layer (i.e., three slabs consisting of input neurons, hidden neurons, and output neurons) are the most widely used delta rule networks. Occasionally, two or more hidden slabs are used for complex mappings between network inputs and outputs.

3.4 A Complete Delta Rule Neural Network Simulator

The complete delta rule neural network simulator listed in this section contains several short test functions. An example execution run of one of these functions, *test2*, is shown after the program listing. The test functions *test2*, *test3*, and *test4* contain respectively two, three and four layers. The screen dump shown in figure 3.7 (from an Apple Macintosh computer) was created while the test function *test3* was executing.

The neurons in this model have activation energies scaled from 0.0 to 1.0; a neuron with an activation energy of 0.0 appears as an empty box on the display. A neuron with a value of 1.0 is shown as a black filled box. Values between 0.0 and 1.0 appear as partially filled boxes (area proportional to value). For example, the four slab2 neurons have values of about 1.0, 0.0, 0.5, and 0.0.

The connection weights are initialized to small random numbers before a neural network is trained. As we will see, the values of these weights change slowly during the training process. When a pattern is input to a neural network (that is, when the

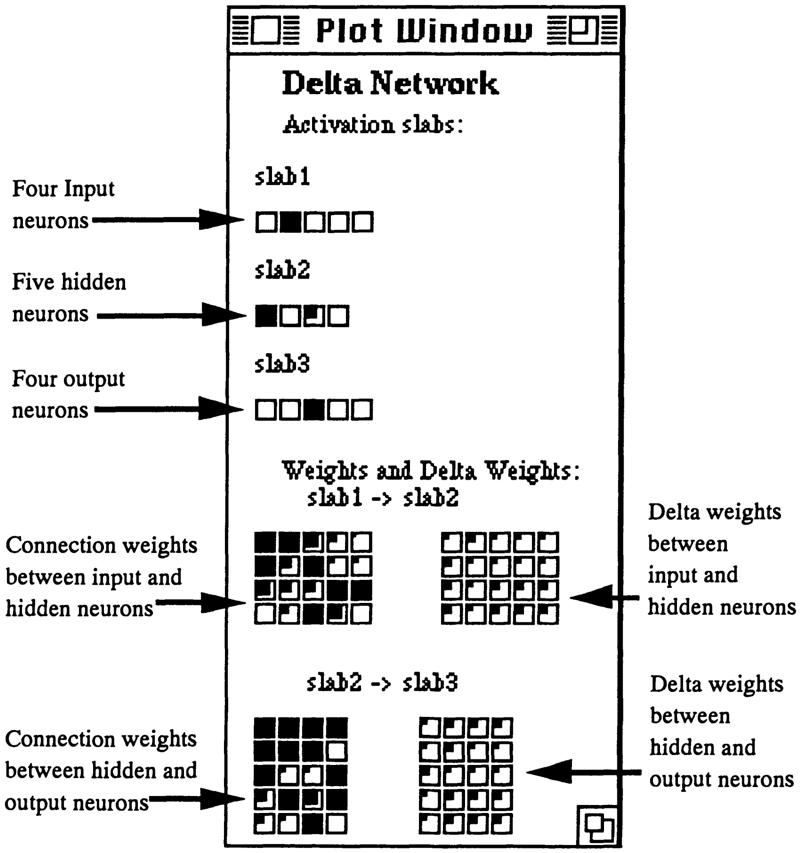

Plot Window
Delta Network
Activation slabs:
slab1
slab2
slab3
Weights and Delta Weights:
slab1 -> slab2
slab2 -> slab3
Four Input neurons
Five hidden neurons
Four output neurons
Connection weights between input and hidden neurons
Connection weights between hidden and output neurons
Delta weights between input and hidden neurons
Delta weights between hidden and output neurons

Figure 3.7. Annotated Execution of Complete Delta Rule Simulator

activation values of the input neurons in slab1 are set equal to the values in an input pattern), the values flow through the connection weights connecting the slab1 and slab2 neurons and cause the activation energy values of the slab2 neurons to be recalculated. In a similar way, the slab2 neuron activation energy values flow through the weights connecting the slab 2 and slab3 neurons affecting the values of the activation values of the slab3 (output) neurons in Figure 3.7.

The magnitude of the two sets of connection-strength weights is shown in the lower left-hand side of the display, while the derivative (rate of change) of the weight sets appears on the right-hand side of the display and is labeled *Delta Weight*. Note that in general, a slab with N neurons is connected to another slab of M neurons with a connection strength weight array of size N by M. For example, in Figure 3.7, the output of the four slab2 neurons is obtained by first multiplying the 5 values in the slab1 array by the 5x4 weight array forming the sum of products one-dimensional array; application of the Sigmoid function converts the one-dimensional sum of products array to the one-dimensional output values for the slab2 neurons. The output for the slab3 neurons is calculated in the same way.

3.4.1 Listing of the Complete Delta Rule Neural Network Simulator

In the following example program, the function *NewDeltaNetwork* is called with a list of integer values, each of which specifies the number of neurons in a simulated layer of neurons. For example, the list *(120 7 5)* would indicate that *NewDeltaNetwork* will create a new neural network with three layers (the length of the list), that the input layer has 120 neurons, the hidden layer has 7 neurons, and the output layer has 5 neurons. Function *NewDeltaNetwork* returns a list of arrays which maintain state information for the neural network. Note that the entire state of a network is maintained in the data structure returned by *NewDeltaNetwork.*

The function *DeltaLearn* is called with two required arguments and two optional arguments. The first argument is the network description list that was returned as the value of function *NewDeltaNetwork.* The second required argument is a list of training data. The first optional argument is an automatic plot flag; if this value is specified as a non-nil value, then the state of the network is plotted in the graphics window during training. The second optional argument identifies a format specification for plotting the layer of input neurons; this option will be used in chapter 6 for recognizing hand-drawn characters (the input neurons will be plotted as a two-dimensional grid so the character being recognized will be seen in the pattern of input neuron activation energies). The returned value from function *DeltaLearn* is the average of the squared error at each output neuron; this returned value will decrease as the network is trained.

The function *DeltaRecall* is called with two required arguments. The first argument is the network description list returned as a value from function *NewDeltaNetwork.* The second argument is a list of input neuron activation energies to propagate through the network. The returned value of function *DeltaRecall* is a list of output neuron activation energies.

The following functions in the listing are not usually called directly from your application programs: *plotActivations* (plots activation energies of a specified layer of neurons at a specified location in the graphics window), *DeltaPlot* (plots the state of the entire network in the graphics window), *Sigmoid* (calculates the value of the Sigmoid function), *dSigmoid* (calculates the value of the derivative of the Sigmoid function), and *frandom* (returns a random number for initializing weight values).

The function *WriteDeltaNetwork* saves the state of a neural network as a disk file. Two required arguments for this function are a disk file name and a network description list. The function *ReadDeltaNetwork* reads a network save file created by function *WriteDeltaNetwork* and returns as its value a network description list.

```
;; File DeltaRule.Lisp
;; Copyright 1990 by Mark Watson

;;;;;;;;;;;; Externally (callable) functions in this file: ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; (NewDeltaNetwork sizeList)
;     Args:   sizeList = list of sizes of slabs. This also defines
;                  the number of slabs in the network.
;                  (e.g., '(10 5 4) ==> a 3-slab network with 10
;                   input neurons, 5 hidden neurons, and 4 output
;                   neurons).
;
;     Returned value = a list describing the network:
;       (nLayers sizeList
;        (activation-array[1] .. activation-array[nLayers])
;        (weight-array[2] .. weight-array[nLayers])
;        (sum-of-products[2] .. sum-of-products[nLayers[nLayers])
;        (back-prop-error[2] .. back-prop-error[nLayers]))
;        (old-delta-weights[2] .. for momentum term

; (DeltaLearn networkList trainingList)
;     Args:    networkList  = list returned from function  NewDeltaNetwork
;           trainingList = a list of lists of training exemplars.
;                    For example, a list might be:
;                    (((0 1) (1 0))  ; first exemplar
;                     ((1 0) (0 1))) ; second exemplar
;                    Note: the inner sub-lists can also be arrays.
;           nIterations  = number of complete training iterations
;
;     Returned value = average error at output neurons for last
;                training cycle.

; (DeltaRecall networkList inputList)
;     Args:    networkList  = list returned from function 'NewDeltaNetwork'
;           inputList    = list OR array of input activation values
;
;     Returned value = list of output neuron values

; (DeltaPlot networkList) ==> plots a network. Must call '(init-plot) first.

; (WriteDeltaNetwork fileName networkList) ==> saves a network to disk


; (ReadDeltaNetwork fileName) ==> returns a network list
```

```
;;;;;;;;;;;;;;;;;; End of list of externally callable functions ;;;;;;;;;;;;

(proclaim '(special eidaList defaultEidaList *delta-default-input-noise-value*
         *delta-rule-debug-flag* ))

;; Define default learning rates for each layer of neurons:

(setq defaultEidaList '(0.5 0.4 0.3 0.2 0.08 0.07))

;; Define the default noise to add to each input neuron:

(setq *delta-default-input-noise-value* 0.08)
(setq *delta-rule-debug-flag* nil)

;;
 ; Create a new delta network:
 ;;

;; alpha = coefficient for new weight change
;; beta  = coefficient for adding in last weight change

(defun NewDeltaNetwork (sizeList &optional (alpha 0.2) (beta 0.8))
  (let ((numLayers (length sizeList))
       (w-list nil)       ; weights
       (dw-list nil)      ; delta weights
       (old-dw-list nil)  ; old delta weights for momentum terms
       (a-list nil)       ; activation values
       (s-list nil)       ; sum of products
       (d-list nil))      ; back propagated deltas

    (setq eidaList defaultEidaList)

     ;;
      ; Initialize storage for activation energy for all slabs:
      ;;
    (setq a-list
         (mapcar
          '(lambda (size) (make-array (list size) :element-type :float))
          sizeList))

    ;;
     ; Initialize storage for sum of products arrays:
     ;;
    (setq s-list
         (mapcar
       '(lambda (size) (make-array (list size) :element-type :float))
       (cdr sizeList)))

;;
; Initialize storage for delta arrays:
;;
(setq d-list
      (mapcar
       '(lambda (size) (make-array (list size) :element-type :float))
       (cdr sizeList)))

;;
; Initialize storage for the weights:
;;
(dotimes (i (- numLayers 1))
  (setq
   w-list
   (cons (list (nth i sizeList) (nth (+ i 1) sizeList)) w-list)))
(setq w-list
      (mapcar
       '(lambda (size) (make-array size :element-type :float))
       (reverse w-list)))

;;
; Initialize the storage for delta weights:
;;
(dotimes (i (- numLayers 1))
  (setq
   dw-list
   (cons (list (nth i sizeList) (nth (+ i 1) sizeList)) dw-list)))
(setq dw-list
      (mapcar
       '(lambda (size) (make-array size :element-type :float))
       (reverse dw-list)))

;;
; Initialize the storage for old delta weights:
;;
(dotimes (i (- numLayers 1))
  (setq
   old-dw-list
   (cons (list (nth i sizeList) (nth (+ i 1) sizeList)) old-dw-list)))
(setq old-dw-list
      (mapcar
       '(lambda (size) (make-array size  :element-type :float
                                :initial-element 0.0))
        (reverse old-dw-list)))

  ;;
  ; Initialize values for all activations:
  ;;
  (mapc
   '(lambda (x)
     (let ((num (array-dimension x 0)))
       (dotimes (n num)
         (setf (aref x n) (frandom 0.01 0.1)))))
   a-list)

  ;;
  ; Initialize values for all weights:
  ;;
  (mapc
   '(lambda (x)
     (let ((numI (array-dimension x 0))
           (numJ (array-dimension x 1)))
       (dotimes (j numJ)
         (dotimes (i numI)
           (setf (aref x i j) (frandom -0.5 0.5))))))
   w-list)
  (list numLayers sizeList a-list s-list w-list dw-list
        d-list old-dw-list alpha beta)))

;;
; Utility function for training a delta rule neural network.
; The first argument is a network definition (as returned from
; NewDeltaNetwork), the second argument is a list of training
; data cases (see the example test functions at the end of this
; file for examples, the third (optional) argument is a flag
; for automatic plotting of the state of the network (activated
; by holding down the mouse button while this function is
; executing, and the fourth (optional) argument is formatting
; information passed to function DeltaPlot to specify plotting
; the input neuron activation values as a two dimentional array
; instead of the (default) format of a one dimensional array.
;;

(defun DeltaLearn (netList trainList
                &optional (autoPlot 'no) (input-format nil))
  (let ((nLayers (car netList))
        (sizeList (cadr netList))
```

```
    (activationList (caddr netList))
    (sumOfProductsList (car (cdddr netList)))
    (weightList (cadr (cdddr netList)))
    (deltaWeightList (caddr (cdddr netList)))
    (deltaList (cadddr (cdddr netList)))
    (oldDeltaWeightList (cadddr (cdddr (cdr netList))))
    (alpha (cadddr (cdddr (cddr netList))))
    (beta (cadddr (cdddr (cdddr netList))))
    (inputs nil)
    (targetOutputs nil)
    (iDimension nil)
    (jDimension nil)
    (iActivationVector nil)
    (jActivationVector nil)
    (n nil)
    (weightArray nil)
    (sumOfProductsArray nil)
    (iDeltaVector nil)
    (jDeltaVector nil)
    (deltaWeightArray nil)
    (oldDeltaWeightArray nil)
    (sum nil)
    (iSumOfProductsArray nil)
    (error nil)
    (outputError 0)
    (delta nil)
    (eida nil)
    (inputNoise 0))
;;
; Zero out deltas:
;;
(dotimes (n (- nLayers 1))
  (let* ((dw (nth n deltaList))
         (len1 (array-dimension dw 0)))
    (dotimes (i len1)
      (setf (aref dw i) 0))))
;;
; Zero out delta weights:
;;
(dotimes (n (- nLayers 1))
  (let* ((dw (nth n deltaWeightList))
         (len1 (array-dimension dw 0))
         (len2 (array-dimension dw 1)))
    (dotimes (i len1)
      (dotimes (j len2)
            (setf (aref dw i j) 0)))))

    (setq inputNoise *delta-default-input-noise-value*)

    ;;
    ; Main loop on training examples:
    ;;
    (dolist (tl trainList)

     (setq inputs (car tl))
     (setq targetOutputs (cadr tl))

     (if *delta-rule-debug-flag*
       (print (list "Current targets:" targetOutputs)))

     (setq iDimension (car sizeList)) ; get the size of the input slab
     (setq iActivationVector (car activationList)) ; get array of input activations
     (dotimes (i iDimension) ; copy training inputs to input slab
       (setf
        (aref iActivationVector i)
        (+ (nth i inputs) (frandom (- inputNoise) inputNoise))))
     ;;
     ; Propagate activation through all of the slabs:
     ;;
     (dotimes (n-1 (- nLayers 1))  ; update layer i to layer flowing to layer j
       (setq n (+ n-1 1))
       (setq jDimension (nth n sizeList)) ; get the size of the j'th layer
       (setq jActivationVector (nth n activationList)) ; activation array for slab j
       (setq weightArray (nth n-1 weightList))
       (setq sumOfProductsArray (nth n-1 sumOfProductsList))
       (dotimes (j jDimension) ; process each neuron in slab j
         (setq sum 0.0) ; init sum of products to zero
         (dotimes (i iDimension) ; to get activation from each neuron in previous slab
           (setq
            sum
            (+ sum (* (aref weightArray i j) (aref iActivationVector i)))))
         (setf (aref sumOfProductsArray j) sum) ; save sum of products
         (setf (aref jActivationVector j) (Sigmoid sum)))
       (setq iDimension jDimension)     ; reset index for next slab pair
       (setq iActivationVector jActivationVector))
     ;;
     ; Activation is  spread through the network and sum of products calculated.
     ; Now modify the weights in the network using back error propagation. Start
     ; by calculating the error signal for each neuron in the output layer:
     ;;
```

```
(setq jDimension (nth (- nLayers 1) sizeList)) ; size of last layer
(setq jActivationVector (nth (- nLayers 1) activationList))
(setq jDeltaVector (nth (- nLayers 2) deltaList))
(setq sumOfProductsArray (nth (- nLayers 2) sumOfProductsList))
(setq outputError 0)
(dotimes (j jDimension)
  (setq delta (- (nth j targetOutputs) (aref jActivationVector j)))
  (setq outputError (+ outputError (abs delta)))
  (setf
   (aref jDeltaVector j)
   (+
    (aref jDeltaVector j)
    (* delta (dSigmoid (aref sumOfProductsArray j))))))
;;
 ; Now calculate the backpropagated error signal for all hidden slabs:
 ;;
(dotimes (nn (- nLayers 2))
  (setq n (- nLayers 3 nn))
  (setq iDimension (nth (+ n 1) sizeList))
  (setq iSumOfProductsArray (nth n sumOfProductsList))
  (setq iDeltaVector (nth n deltaList))
  (dotimes (i iDimension)
    (setf (aref iDeltaVector i) 0.0))
  (setq weightArray (nth (+ n 1) weightList))
  (dotimes (i iDimension)
    (setq error 0.0)
    (dotimes (j jDimension)
      (setq error (+ error (* (aref jDeltaVector j) (aref weightArray i j)))))
    (setf
     (aref iDeltaVector i)
     (+
      (aref iDeltaVector i)
      (* error (dSigmoid (aref iSumOfProductsArray i))))))
  (setq jDimension iDimension)
  (setq jDeltaVector iDeltaVector))

;;
 ; Update all delta weights in the network:
 ;;
(setq iDimension (car sizeList))
(dotimes (n (- nLayers 1))
  (setq iActivationVector (nth n activationList))
  (setq jDimension (nth (+ n 1) sizeList))
  (setq jDeltaVector (nth n deltaList))
  (setq deltaWeightArray (nth n deltaWeightList))
```

```
      (setq weightArray (nth n weightList))
      (setq eida (nth n eidaList))

      (dotimes (j jDimension)
        (dotimes (i iDimension)
          (setq delta (* eida (aref jDeltaVector j) (aref iActivationVector i)))
          (setf
           (aref DeltaWeightArray i j)
           (+ (aref DeltaWeightArray i j) delta)))) ; remember delta weight change

      (setq iDimension jDimension))

    (if (not (equal autoPlot 'no))
      (DeltaPlot netList input-format)))

  ;;
   ; Update all weights in the network:
   ;;
  (setq iDimension (car sizeList))
  (dotimes (n (- nLayers 1))
    (setq iActivationVector (nth n activationList))
    (setq jDimension (nth (+ n 1) sizeList))
    (setq jDeltaVector (nth n deltaList))
    (setq deltaWeightArray (nth n deltaWeightList))
    (setq oldDeltaWeightArray (nth n oldDeltaWeightList))
    (setq weightArray (nth n weightList))
    (dotimes (j jDimension)
      (dotimes (i iDimension)
        (setf
         (aref weightArray i j)
         (+ (aref weightArray i j)
           (* alpha (aref deltaWeightArray i j))
           (* beta  (aref oldDeltaWeightArray i j))))
         (setf (aref oldDeltaWeightArray i j) ; save current delta weights
              (aref deltaWeightArray i j)))) ; ...for next momentum term.
    (setq iDimension jDimension))

  (/ outputError jDimension)))

;;
; Utility for using a trained neural network in the recall mode.
; The first argument to this function is a network definition (as
; returned from NewDeltaNetwork) and the second argument is a list
; of input neuron activation values to drive through the network.
;;
```

```
(defun DeltaRecall (netList inputs)
  (let ((nLayers (car netList))
        (sizeList (cadr netList))
        (activationList (caddr netList))
        (weightList (cadr (cdddr netList)))
        (iDimension nil)
        (jDimension nil)
        (iActivationVector nil)
        (jActivationVector nil)
        (n nil)
        (weightArray nil)
        (returnList nil)
        (sum nil))
    (setq iDimension (car sizeList)) ; get the size of the input slab
    (setq iActivationVector (car activationList)) ; get array of input activations
    (dotimes (i iDimension) ; copy training inputs to input slab
      (setf (aref iActivationVector i) (nth i inputs)))
    (dotimes (n-1 (- nLayers 1))  ; update layer j to layer i
      (setq n (+ n-1 1))
      (setq jDimension (nth n sizeList)) ; get the size of the j'th layer
      (setq jActivationVector (nth n activationList)) ; activation array for slab j
      (setq weightArray (nth n-1 weightList))
      (dotimes (j jDimension) ; process each neuron in slab j
        (setq sum 0.0) ; init sum of products to zero
        (dotimes (i iDimension) ; to get activation from each neuron in previous slab
          (setq
           sum
           (+ sum (* (aref weightArray i j) (aref iActivationVector i)))))
        (if *delta-rule-debug-flag*
          (print (list "sum=" sum)))
        (setf (aref jActivationVector j) (Sigmoid sum)))
      (setq iDimension jDimension) ; get ready for next slab pair
      (setq iActivationVector jActivationVector))
    (dotimes (j jDimension)
      (setq returnList (append returnList (list (aref jActivationVector j)))))
    returnList))

;;
; Utilities to plot a network
;;

(defun plotActivations (title x y data dmin dmax &optional (input-format nil))
  (let ((size (array-dimension data 0)) (ypos 0) (xpos x))
    (plot-string x (- y 10) title)
    (if input-format
    (let ((x-size (car input-format)) (y-size (cadr input-format)))
      (dotimes (yy y-size)
        (dotimes (xx x-size)
          (plot-size-rect
           (truncate (+ xpos (* 9 xx)))
           (truncate (+ ypos (* 9 yy) 20))
           8 8
           (truncate
            (*
             (/
              (- (aref data (+ (* yy x-size) xx)) dmin)
              (- dmax dmin)) 8)))))))
    (dotimes (i size)
      (if (< size 20)
        (setq ypos y xpos (+ x (* i 9)))
        (if (< i (/ size 2))
          (setq ypos (- y 7) xpos (+ x (* i 9)))
          (setq ypos (+ y 2)  xpos (+ x (* (- i (/ size 2)) 9)))))
      (plot-size-rect
       (truncate xpos) (truncate ypos) 8 8
       (truncate (* (/  (- (aref data i) dmin) (- dmax dmin)) 8)))
      (plot-frame-rect (truncate xpos) (truncate ypos) 8 8)))))

(defun plotWeights (title x y data dmin dmax deltaWeights)
  (let ((Xsize (array-dimension data 0))
        (YSize (array-dimension data 1)))
   (if (< (* Xsize Ysize) 200)     ;; don't try to plot very large weight sets
     (progn
      (plot-string (+ x 20) (- y 10) title)
      (dotimes (i Xsize)
        (dotimes (j Ysize)
          (plot-size-rect
           (+ x (* i 9)) (+ y (* j 9)) 8 8
           (truncate (* (/  (- (aref data i j) dmin) (- dmax dmin)) 8)))
          (plot-frame-rect (+ x (* i 9)) (+ y (* j 9)) 8 8)
          (plot-size-rect
           (+ (* Xsize 8) 30 x (* i 9)) (+ y (* j 9)) 8 8
           (truncate (* (/  (- (aref deltaWeights i j) -.05) (- .05 -.05)) 8)))
          (plot-frame-rect (+ (* Xsize 8) 30 x (* i 9)) (+ y (* j 9)) 8 8)))))))

(defun DeltaPlot (netList input-format &optional (plotOnlyIfMouseClick 'yes))
  (if (or
       (not (equal plotOnlyIfMouseClick 'yes))
       (plot-mouse-down))  ;; only plot if mouse is clicked !
   (let ((nLayers (car netList))
    (sizeList (cadr netList))
    (activationList (caddr netList))
    (sumOfProductsList (car (cdddr netList)))
    (weightList (cadr (cdddr netList)))
    (deltaWeightList (caddr (cdddr netList)))
    (minScale -0.3)
    (maxScale 0.3)
    (y-start 0))
 (show-plot)
 (plot-string-bold 20 15 "Delta Network")
 (plot-string 20 30 "Activation slabs:")
 (plotActivations "slab1"
           10
           60
           (nth 0 ActivationList)
           -0.5
           0.5
           input-format)
 (if input-format
   (setq y-start (+ 30 (* (cadr input-format) 11)))
   (setq y-start 60))
 (dotimes (n-1 (- nLayers 1))
   (setq n (+ n-1 1))
   (if (equal n (- nLayers 1))
     (setq minScale -0.1 maxScale 0.1)) ; scale up output display
   (plotActivations
    (nth n '("slab1" "slab2" "slab3" "slab4" "slab5")) ; title
    10 ; x location for subplot
    (+ y-start (* n 35)) ; y location for subplot
    (nth n ActivationList) ; data to plot as gray scale
    minScale
    maxScale
    nil))
 (if (< nLayers 4)
   (progn
     (setq y-start (+ y-start 100))
     (plot-string 20 y-start "Weights and Delta Weights:")
     (setq y-start (+ y-start 20))
     (dotimes (n (- nLayers 1))
       (plotWeights
        (nth n '("slab1 -> slab2"  "slab2 -> slab3" "slab3 -> slab4"))
        10 (+ y-start (* n 70)) ; x,y position of subplot
       (nth n WeightList)
       -1.0 1.0
       (nth n deltaWeightList)))))))
```

```
;;
 ; Calculate Sigmoid and derivative of Sigmoid functions:
 ;;

(defun Sigmoid (x)
  (/ 1.0 (+ 1.0 (exp (- x)))))

(defun dSigmoid (x)
  (let ((temp (Sigmoid x)))
    (* temp (- 1.0 temp))))


;;
 ; Generate floating point random numbers:
 ;;

(defun frandom (low high)
  (let ((range (- high low)))
    (+ (* (/ (random 1000) 1000.0) range) low)))


;;
 ; Save a Delta Network to a disk file:
 ;;

(defun WriteDeltaNetwork (fileName netList)
  (let ((fileStream (open fileName :direction :output))
        (nLayers (car netList))
        (sizeList (cadr netList))
        (activationList (caddr netList))
        (weightList (cadr (cdddr netList)))
        (deltaWeightList (caddr (cdddr netList)))
        (oldDeltaWeightList (car (cddddr (cdddr netList))))
        (alpha (cadr (cddddr (cdddr netList))))
        (beta (caddr (cddddr (cdddr netList)))))
    ;;
     ; Write out header:
     ;;
    (print nLayers fileStream)
    (print sizeList fileStream)
    ;;
     ; Write out activations:
     ;;
    (dotimes (n nLayers)
      (dotimes (i (nth n sizeList))
      (print (aref (nth n activationList) i) fileStream)))
    ;;
    ; Write out weights:
    ;;
    (dotimes (n (- nLayers 1))
      (let ((w (nth n weightList)))
        (dotimes (i (array-dimension w 0))
          (dotimes (j (array-dimension w 1))
            (print (aref w i j) fileStream)))))
    ;;
    ; Write out delta weights:
    ;;
    (dotimes (n (- nLayers 1))
      (let ((w (nth n deltaWeightList)))
        (dotimes (i (array-dimension w 0))
          (dotimes (j (array-dimension w 1))
            (print (aref w i j) fileStream)))))
    ;;
    ; Write out old delta weights:
    ;;
    (dotimes (n (- nLayers 1))
      (let ((w (nth n oldDeltaWeightList)))
        (dotimes (i (array-dimension w 0))
          (dotimes (j (array-dimension w 1))
            (print (aref w i j) fileStream)))))
    ;;
    ; Write alpha, beta terms (used for momentum):
    ;;
    (print alpha fileStream)
    (print beta fileStream)

    (close fileStream)))

;;
; Read a delta detwork from a disk file:
;;

(defun ReadDeltaNetwork (fileName)
  (let ((fileStream (open fileName :direction :input))
        (numLayers nil)
        (sizeList nil)
        (a-list nil)
        (s-list nil)
        (w-list nil)
        (dw-list nil)
        (old-dw-list)
        (d-list nil)
        alpha beta)

  ;;
   ; Read in header:
   ;;
  (setq numLayers (read fileStream))
  (setq sizeList (read fileStream))

  ;;
   ; Allocate array storage:
   ;;
  (setq a-list
       (mapcar
        '(lambda (size) (make-array (list size)))
        sizeList))
  (setq s-list
       (mapcar
        '(lambda (size) (make-array (list size)))
        (cdr sizeList)))
  (setq d-list
       (mapcar
        '(lambda (size) (make-array (list size)))
        (cdr sizeList)))
  (dotimes (i (- numLayers 1))
    (setq
     w-list
     (cons (list (nth i sizeList) (nth (+ i 1) sizeList)) w-list)))
  (setq w-list
       (mapcar
        '(lambda (size) (make-array size))
        (reverse w-list)))

  (dotimes (i (- numLayers 1))
    (setq
     dw-list
     (cons (list (nth i sizeList) (nth (+ i 1) sizeList)) dw-list)))

  (dotimes (i (- numLayers 1))
    (setq
     old-dw-list
     (cons (list (nth i sizeList) (nth (+ i 1) sizeList)) old-dw-list)))

  (setq dw-list
      (mapcar
       '(lambda (size) (make-array size))
       (reverse dw-list)))

 (setq old-dw-list
      (mapcar
       '(lambda (size) (make-array size))
       (reverse old-dw-list)))

;;
; Read in activations:
;;
(dotimes (n numLayers)
  (dotimes (i (nth n sizeList))
    (setf (aref (nth n a-list) i) (read fileStream))))

;;
; Read in weights:
;;
(dotimes (n (- numLayers 1))
  (let ((w (nth n w-list)))
    (dotimes (i (array-dimension w 0))
      (dotimes (j (array-dimension w 1))
        (setf (aref w i j) (read fileStream))))))

;;
; Read in delta weights:
;;
(dotimes (n (- numLayers 1))
  (let ((w (nth n dw-list)))
    (dotimes (i (array-dimension w 0))
      (dotimes (j (array-dimension w 1))
        (setf (aref w i j) (read fileStream))))))

;;
; Read in old delta weights:
;;
(dotimes (n (- numLayers 1))
  (let ((w (nth n old-dw-list)))
    (dotimes (i (array-dimension w 0))
      (dotimes (j (array-dimension w 1))
        (setf (aref w i j) (read fileStream))))))

(setq alpha (read fileStream)
     beta (read fileStream))
```

```
   (close fileStream)
   (list numLayers sizeList a-list s-list w-list dw-list d-list
        old-dw-list alpha beta)))

;;
; Throw away test functions for two, three and four layer networks:
;;

(proclaim '(special temp))  ; temp will be used to hold the network data

(defun test2 (&optional (restart 'yes) &aux RMSerror)
  (if
   (equal  restart 'yes)
    (setq temp (newdeltanetwork '(2 2)))) ; specify a two layer network (2x2)
  (dotimes (ii 10000)
    (setq
     RMSerror
     (deltalearn
      temp
      '(((1 0) (0 1))
        ((0 1) (1 0)))
      'yes)) ; autoplot mode
    (if (equal (mod ii 50) 0) ;; print out every 50 cycles
      (progn
        (princ "....training cycle \#")
        (princ ii)
        (princ " RMS error = ")
        (princ RMSerror)
        (terpri)))))

(defun test3 (&optional (restart 'yes) &aux RMSerror) ; three layer network
  (if
    (equal restart 'yes)
    (setq temp (newdeltanetwork '(5 4 5))))
  (dotimes (ii 10000)
    (setq
     RMSerror
     (deltalearn
      temp
      '(((1 0 0 0 0) (0 1 0 0 0))
        ((0 1 0 0 0) (0 0 1 0 0))
        ((0 0 1 0 0) (0 0 0 1 0))
        ((0 0 0 1 0) (0 0 0 0 1))
        ((0 0 0 0 1) (1 0 0 0 0)))
    'yes)) ; autoplot mode
   (if (equal (mod ii 50) 0) ;; print out every 50 cycles
    (progn
     (princ "....training cycle \#")
     (princ ii)
     (princ " RMS error = ")
     (princ RMSerror)
     (terpri)))))

(defun test4 (&optional (restart 'yes) &aux RMSerror)  ; four layer network
  (if
   (equal restart 'yes)
   (setq temp (newdeltanetwork '(4 5 5 4))))
  (dotimes (ii 10000)
   (setq
    RMSerror
    (deltalearn
     temp
     '(((1 0 0 0) (0 1 0 0))
       ((0 1 0 0) (0 0 1 0))
       ((0 0 1 0) (0 0 0 1))
       ((0 0 0 1) (1 0 0 0)))
     'yes)) ; autoplot mode
   (if (equal (mod ii 50) 0) ;; print out every 50 cycles
    (progn
     (princ "....training cycle \#")
     (princ ii)
     (princ " RMS error = ")
     (princ RMSerror)
     (terpri)))))
```

3.4.2 Running the Complete Delta Rule Simulator

Before running any of the sample test functions (i.e., *test2*, *test3*, or *test4*) at the end of the above listing, remember to initialize a graphics window on your system by loading the device-independent plotting library and executing the function *init-plot*. Note: holding down the mouse button while the sample test programs are running will update the display in the graphics window if *DeltaLearn* is called in the autoplot mode (third argument = *yes*).

The following example was obtained after executing the function *test2* (text in a **bold typeface** is user input in the Lisp listener window, while text in normal typeface is the program output):

```
? (test2)
....training cycle #0 RMS error = 0.5009396286537473
```

```
....training cycle #50 RMS error = 0.2816511201933224
....training cycle #100 RMS error = 0.2070144326479426
....training cycle #150 RMS error = 0.1594081687333126
....training cycle #200 RMS error = 0.1078002482272793
nil
? (deltaRecall temp '(0 1))     ;; global variable temp defined in test2
(0.8829278986163372
 0.1226715742976734)
? (deltaRecall temp '(1 0))
(8.036152662989635E-2
 0.9165891912418624)
? (test2 'no) ;; retrain the same network stored in the list temp. no = no restart
....training cycle #0 RMS error = 0.1514774882974489
....training cycle #50 RMS error = 0.1110860207023915
....training cycle #100 RMS error = 7.343453188164262E-2
....training cycle #150 RMS error = 8.759386766194083E-2
....training cycle #200 RMS error = 8.340545162724894E-2
nil
?(deltaRecall temp '(0 1)) ;; results after more training will be more accurate!!
(0.9222244615723154
 7.938495304694376E-2)
?(deltaRecall temp '(1 0)) ;; results after more training will be more accurate!!
(5.520507817649352E-2
 0.9437721804513303)
?
```

The LISP variable *temp* is the returned value from the function *NewDeltaNetwork* which creates a new delta rule neural network in our LISP environment; *NewDeltaNetwork* takes one argument: a list of integers whose length specifies the number of layers (or slabs) in the network, the values in the list specifying the size of each layer of neurons in the network. By performing a Common LISP "pretty print" (pprint) operation on the variable *temp*, you can inspect the numerical values for each activation value and weight.

The function *DeltaRecall* can be used to test a network; its two arguments are a neural network data definition (e.g., the data structure stored in the variable *temp* in the above example) and a list of values for the input neurons. The function *DeltaRecall* returns a list of output neuron activation values.

This small example function *test2* shows how to train a two-layer neural network to map the input pattern (1.0, 0.0) into the pattern (0.0, 1.0), and to map the pattern (0.0, 1.0) into the output pattern (1.0, 0.0). If this example program was also given the training data to map the pattern (1.0, 1.0) mapping into the output pattern (0.0, 0.0) and to map the input pattern (0.0, 0.0) into the output pattern (1.0, 1.0), then the neural network would have been trained to take an input pattern and map it to its complement (or opposite) value.

Function *test3* trains a larger network with three layers (input, hidden and output). Function *test4* trains a four-layer network that uses two hidden layers of neurons between the input and output layers.

3.4.3 Preparing Training Data for the Delta Rule Simulator

There are several techniques that can contribute to the successful training of a delta rule network: adding random noise to the input training data, using a large set of training data, performing algorithmic preprocessing of training data when possible, and allowing the network to learn by processing the training data set thousands of times with a small learning rate.

3.4.3.1 Addition of Random Noise

The addition of random noise facilitates training a delta rule neural network for at least two reasons: simulation of a larger training data set and allowing a network to break out of local minima during early training. Sometimes a network will develop a weight set which minimizes the error for (i.e., can recognize) most of the input training vectors, but fails to learn all input patterns. Since delta rule learning is a steepest-descent learning process, it can sometimes (but not often) get stuck in local minima; you have seen the same phenomenon in high school math when using Newton's method for finding the roots for polynomials: if you start in the wrong position, the iteration to calculate a root "gets stuck" in a local minima.

3.4.3.2 Large Training Data Set

It is almost impossible to have too much training data for a neural network. There is usually noise in training data sets. A network which is forced to learn from a large set of examples generalizes the important features in the data and ignores noise and occasional errors in the training input data. A common error is to train a neural network with only a few examples; when the network tries to match a new pattern that is very similar to one of the training inputs, the desired match may not occur. A simple example makes this clear: if you only show a small child two houses and they are both blue, the child will assume that a house has to be blue; if the child is shown hundreds of houses, she will learn to make generalizations based on what features are really important in differentiating a house from a barn or office building.

When many training examples are available, randomly choose a small percentage of the training examples and train the network with this small initial set. After a short initial training interval, add the remaining training examples. This technique will often reduce total training time when dealing with many training examples.

3.4.3.3 Algorithmic Preprocessing

Often training can be greatly facilitated by appropriate preprocessing of input data. Good examples of this processing are the example in chapter 5 for speech recognition where spectral components of raw speech data are calculated as input to the neural network, and the example in chapter 6 for recognition of handwritten characters where spatial registration and resizing of characters for input to a neural network facilitates training the network. Do not use neural networks for computational tasks which are more easily performed using other methods.

3.4.3.4 Use a Small Learning Rate

Use a small learning rate when training networks. The variables ****learning-rate**** (in the short example program in section 3.3.1) and *defaultEidaList* (in the complete delta rule simulator) are used to set the learning rate. Be patient when training neural networks; set up overnight training runs to train networks while you are not using your computer for other work. If your computer does not have hardware for floating point arithmetic, you should experiment with small networks as larger networks could potentially take days to train.

Figure 3.8 shows the decreasing error rate when the test function *test4* was executed. Function *DeltaLearn* returns the average root-mean-square (RMS) error for the output neurons; this is a measure of how closely the network's output pattern matches the training output pattern. DeltaLearn was called 5000 times in function *test4* and the RMS error is plotted against number of calls to *DeltaLearn* in Figure 3.8. If the learning rate is too large, the output error will not continue to decrease. Do not expect the output error to monotonically decrease; you will see some fluctuation, like that in Figure 3.8.

3.5 Projects

1. Using the neural network simulator provided at the end of this chapter, try to train a neural network with 4 input neurons and 4 output neurons to show the complement of the input neurons in the output neurons. Try networks with the following structure (not all of which will solve this mapping problem):

    - 4 input and 4 output neurons (no hidden layer)
    - 4 input, 1 hidden, and 4 output neurons
    - 4 input, 2 hidden, and 4 output neurons
    - 4 input, 3 hidden, and 4 output neurons
    - 4 input, 4 hidden, and 4 output neurons
    - 4 input, 20 hidden, and 4 output neurons

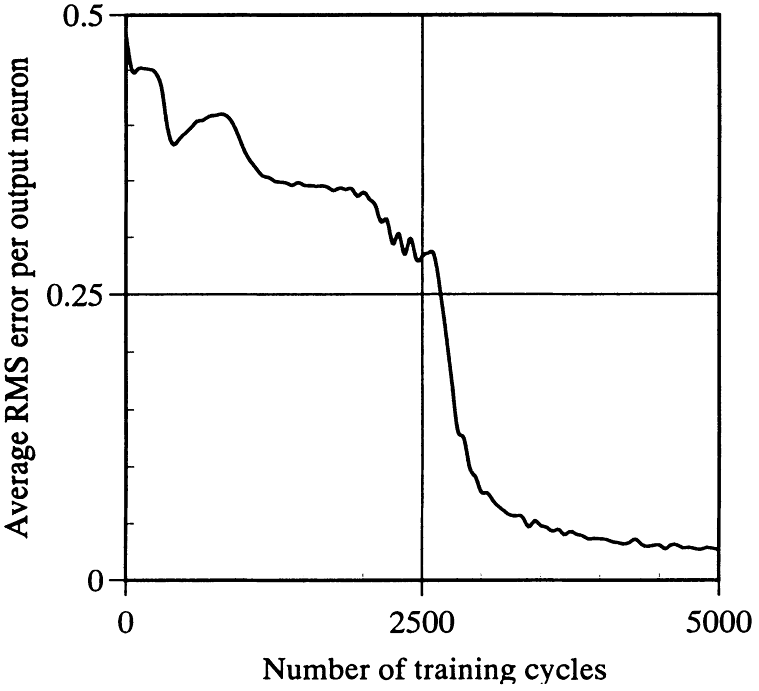


Figure 3.8. The average error for each output neuron decreases during training

Answer the following questions:

- How many unique inputs do I have?
- What is the minimum number of hidden neurons that the network can have and still theoretically learn all of the input patterns?

NOTE: If you computer does not have hardware to support floating point arithmetic, scale down this problem to 3 input and 3 output neurons to reduce training time. The time required to train a network increases with the size of the network. Reducing the number of input and output neurons in a network will thus decrease the required training time.

2. Collect training data for any application that interests you. Possible ideas are: raw data for weather prediction, attributes of winning race horses, and maintenance data and repair records for automobiles. You will need to scale the input training data to a dynamic range from 0.0 to 1.0. Since delta rule neural networks require supervised training, you will need to categorize the training data into output categories. The input layer of your neural network will have one input neuron for each input parameter. The output

layer will have one neuron for each output category that you create for your data. After training your network, use new data to test its accuracy in categorizing this new data which it has not seen before.

Hints: Start with a 2 layer (input and output, no hidden layers) since the training will not take as long as networks with hidden layers. Then add one hidden layer, and test it with new data to evaluate its accuracy relative to the simpler 2 layer network. Remember, complicated problems require a large number of input/output training patterns.

4
Pattern Recognition Using Hopfield Neural Networks

In the last chapter, we saw how a generalized delta rule (backwards error propagation) network could slowly learn to recognize a series of patterns. The delta rule network adapted slowly while training repetitively on a set of examples (sometimes a set of training examples passes through the network over 100,000 times!).

In this chapter we will study an example program which implements a class of networks commonly known as Hopfield networks, after John Hopfield, who is a member of the National Academy of Science and who has helped popularize neural networks. Unlike delta rule networks, the training time for Hopfield networks is linear with the number of training examples and is predictable. Hopfield neural networks are appropriate for solving application problems requiring associative memory recall, such as retrieving digitized images from an image database when the recall key is a low-resolution or partially destroyed copy of the original. For example, if your company issued picture identification cards to employees, these photographs could be stored in a Hopfield neural network and used by a video security system to admit personnel into you building; a Hopfield network can recognize video images even if they are incomplete or taken at a slightly different angle than the original training image. Training Hopfield networks is simple both in the ease of programming and in reduced machine resources. The ability to recall patterns which are partially destroyed by noise is demonstrated by the example program presented in this chapter.

4.1 Mathematical Model

Hopfield neural networks are simple to implement in software and to embed in larger application programs. In the following design description for Hopfield network implementation, the symbol *I* refers to an input image to be stored in a Hopfield network. If the input patterns to the network are vectors of length *N*, the weight matrix for the Hopfield neural network is simply a *N*x*N* autocorrelation matrix of all the input patterns. On associative recall of a vector of length *N*, the training input pattern which most closely matches the new input is returned. The input image in the example program is a one-dimensional image of alternating light and dark patches of differing widths. The following analysis and software is equally valid for higher dimensional images: for example, if you are storing 60x40 pixel (two-dimensional) images in a Hopfield network, the input training image *I* can be treated as a 2400 element one-dimensional array.

The weight matrix for a Hopfield neural network is calculated as:

$$W_{i,j} = \sum_{p=1}^{NT} {I^{P}}^{T}_{i} I^{P}_{j}$$

Where:

I^{P} = training input number p

I^{P^T} = training input number p, transposed

NT = the number of pixels in a training image

The output of the Hopfield network is simply given by:

$$O_i = clip\left(\sum_{j \neq i}^{NT} W_{i,j} I_j \right)$$

$$clip(x) = \begin{cases} 1 & if\ (x > 0) \\ -1 & if\ (x \leq 0) \end{cases}$$

I_j = element number j of the recalled input

The function *clip* returns a value of 1 if its argument is greater than zero, and returns -1 if its argument is less than or equal to zero. The weight matrix is also known as an autocorrelation matrix, which is defined for a matrix as the product of a matrix with its transposition (i.e., reversing its row and column indices).

4.2 Listing of the Hopfield Simulator

The following sample program implements the equations for calculating Hopfield neural network connection strength weights presented in section 4.1. The first function defined in this listing (*Hopfield-Init*) sets up global variables for the number of input neurons, the number of training examples, a list containing the training data, storage for the input neuron activation values, and storage for the connection-strength weights. The weights (stored in variable **Hopfield-Weights**) are calculated as the autocorrelation of the input patterns.

The second function defined in the following listing is *HopfieldNetRecall* which has two arguments: a list of data arrays specifying the state of the neural network, and the number of relaxation iterations to use (5 is a good value) in recognizing the pattern in the **inputCells** component of the network description list.

```
;;
; Common LISP Hopfield Network for associative memory recall.
;           Copyright 1989 by Mark Watson
;
; The following function is passed a list
; of lists of training examples.  A Hopfield
; network is built in memory to recognize
; patterns similar to the training examples.
; This function returns a list containing the
; data defining a Hopfield neural network.
; All data for the network is encapsulated
; in the list returned by this function, so
; multiple Hopfield neural networks can be
; used in an application program.
;;

(defun Hopfield-Init (training-data
                &aux temp *num-inputs* *num-training-examples*
                   *training-list* *inputCells* *tempStorage*
                   *HopfieldWeights*)

 ;;
 ; Allocate global arrays for data storage:
 ;;

 (setq *num-inputs* (length (car training-data)))
 (setq *num-training-examples* (length training-data))

 (setq *training-list* (make-array (list *num-training-examples* *num-inputs*)))
 (setq *inputCells* (make-array (list *num-inputs*)))
 (setq *tempStorage* (make-array (list *num-inputs*)))
 (setq *HopfieldWeights* (make-array (list *num-inputs* *num-inputs*)))

 ;;
 ;  Copy training data:
 ;;

 (dotimes (j *num-training-examples*)
  (dotimes (i *num-inputs*)
   (setf
    (aref *training-list* j i)
    (nth i (nth j training-data)))))
 ;;
 ; Utility to adjust data values to values of -1.0 or +1.0 only:
 ;;
```

```
(defun adjustInput (value)  ;; this function is lexically scoped
  (if (< value 0.1)
    -1.0
    +1.0))


;;
 ;  Adjust training data to -1.0 and +1.0 values:
 ;;

(dotimes (i *num-inputs*)
  (dotimes (n *num-training-examples*)
    (setf
     (aref *training-list* n i)
     (adjustInput (aref *training-list* n i)))))

(dotimes (i *num-inputs*)
  (dotimes (j *num-inputs*)
    (setf (aref *HopfieldWeights* i j) 0)))

;;
 ; Calculate the autocorrelation weight matrix from the input test patterns:
 ;;

(dotimes (j-1 (- *num-inputs* 1))
  (let ((j (+ j-1 1)))
    (dotimes (i j)
      (dotimes (s *num-training-examples*)
        (setq temp
             (truncate
              (+
               (*  ;; 2 if's truncate values to -1 or 1:
                (adjustInput (aref *training-list* s i))
                (adjustInput (aref *training-list* s j)))
               (aref *HopfieldWeights* i j))))
        (setf (aref *HopfieldWeights* i j) temp)
        (setf (aref *HopfieldWeights* j i) temp)))))
(dotimes (i *num-inputs*)
  (setf (aref *tempStorage* i) 0)
  (dotimes (j i)
    (setf (aref *tempStorage* i)
         (+ (aref *tempStorage* i) (aref *HopfieldWeights* i j)))))
;;
 ; Return the value of the Hopfield network data object:
 ;;
 (list
  *num-inputs* *num-training-examples* *training-list*
  *inputCells* *tempStorage* *HopfieldWeights*))


;;
; Iterate the network to let it settle in a stable pattern
; which will (we hope) be the original training pattern
; most closely resembling the noisy test pattern:
;;

(defun HopfieldNetRecall (aHopfieldNetwork numberOfIterations)

  ; Define a block of code in which all data components of our
  ; test Hopfield neural network are defined as lexically scoped:
  (let ((*num-inputs* (nth 0 aHopfieldNetwork))
        (*num-training-examples* (nth 1 aHopfieldNetwork))
        (*training-list* (nth 2 aHopfieldNetwork))
        (*inputCells* (nth 3 aHopfieldNetwork))
        (*tempStorage* (nth 4 aHopfieldNetwork))
        (*HopfieldWeights* (nth 5 aHopfieldNetwork)))
  ;;
  ; Calculate a change in energy from old input
  ; values and the autocorrelation weight matrix:
  ;;

 (defun deltaEnergy (k y &aux (temp 0.0))  ;; this function is lexically scoped
  (dotimes (j *num-inputs*)
   (setq temp (+ temp (* (aref *HopfieldWeights* k j) (aref y j)))))
  (- (* 2.0 temp) (aref *tempStorage* k)))

 ;;
 ; Main code for HopfieldNetRecall:
 ;;

 (dotimes (ii numberOfIterations)
  (dotimes (i *num-inputs*)
   (setf (aref *inputCells* i)
        (if (> (deltaEnergy i *inputCells*) 0)
            1
            0))))))
```

```
;;
;  Throwaway test code: to run, execute: (test)
;
;  You can edit the following value plot-size to
;  change the size of the cells in the output graph:
;;

(setq plot-size 8)
(setq plot-size+1 (+ plot-size 1))

;;
; Main test program: execute (test) in your Lisp listener window
;;

(defun test (&aux aHopfieldNetwork)
 (init-plot "Hopfield pattern classifier" 280 160)
 (let ((tdata '( ;; sample sine wave data with different periods:
            (1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 0 0 1 1 0 0 0)
            (0 1 1 0 0 0 0 0 1 0 0 1 0 0 0 0 0 1 0 0 0 1 1 0 0 1 0)
            (0 0 0 1 1 0 0 0 0 0 0 1 1 0 0 0 0 0 1 1 0 0 1 1 0 1 1))))

  ; Set up network:
  (setq aHopfieldNetwork (Hopfield-Init tdata))

  ;;
  ; Define a block of code in which all data components of our
  ; test Hopfield neural network are defined as lexically scoped
  ; variables to make writing the test code easier:
  ;;

  (let ((*num-inputs* (nth 0 aHopfieldNetwork))
       (*num-training-examples*  (nth 1 aHopfieldNetwork))
       (*training-list* (nth 2 aHopfieldNetwork))
       (*inputCells* (nth 3 aHopfieldNetwork))
       (*tempStorage* (nth 4 aHopfieldNetwork))
       (*HopfieldWeights* (nth 5 aHopfieldNetwork)))

   ;;
   ; Define debug functions for plotting data values. Note that
   ; these plot functions are all lexically scoped inside function test.
   ;;

   (defun plotExemplar (row &aux (dmin 0.0) (dmax 1.0) (x 20) (y 30))
    (let ((Xsize (array-dimension *training-list* 0))
         (YSize (array-dimension *training-list* 1)))
    (plot-string-italic (+ x 20) (- y 10) "Original Training Exemplar")
    (dotimes (j Ysize)
      (plot-fill-rect
       (+ x (* j plot-size+1)) y plot-size plot-size
       (truncate (* (/  (- (aref *training-list* row j) dmin) (- dmax dmin)) 5)))
      (plot-frame-rect (+ x (* j plot-size+1)) y plot-size plot-size))))


(defun plot-*original-inputCells* (&aux (dmin 0.0) (dmax 1.0) (x 20) (y 70))
  (let ((Xsize (array-dimension *inputCells* 0)))
    (plot-string-italic (+ x 20) (- y 10) "Scrambled Inputs")
    (dotimes (j Xsize)
      (plot-fill-rect
       (+ x (* j plot-size+1)) y plot-size plot-size
       (truncate (* (/  (- (aref *inputCells* j) dmin) (- dmax dmin)) 5)))
      (plot-frame-rect (+ x (* j plot-size+1)) y plot-size plot-size))))

(defun plot-*inputCells* (&aux (dmin 0.0) (dmax 1.0) (x 20) (y 110))
  (let ((Xsize (array-dimension *inputCells* 0)))
    (plot-string-italic (+ x 20) (- y 10) "Reconstructed Inputs")
    (dotimes (j Xsize)
      (plot-fill-rect
       (+ x (* j plot-size+1)) y plot-size plot-size
       (truncate (* (/  (- (aref *inputCells* j) dmin) (- dmax dmin)) 5)))
      (plot-frame-rect (+ x (* j plot-size+1)) y plot-size plot-size))))

;;
  ; Define a utility function to scramble training inputs
  ; for use in testing the Hopfield network.  This function is lexically scoped:
 ;;

(defun modifyInput (arrSize arr)  ;; modify input array for testing
  (dotimes (i arrSize)
    (if (< (random 50) 5)
      (if (> (aref arr i) 0)
        (setf (aref arr i) -1)
        (setf (aref arr i) 1)))))

;;
  ; Test network on training data that is randomly modified:
 ;;

(dotimes (iter 1000) ;; user can break out early
  (dotimes (s *num-training-examples*)
    (dotimes (i *num-inputs*)
    (setf (aref *inputCells* i) (aref *training-list* s i)))
   (plotExemplar s)
   (modifyInput *num-inputs* *inputCells*)
   (plot-*original-inputCells*)

   ;;
   ; NOTE:  usually you will want to call function HopfieldNetRecall
   ;        with a second argument of 5 (to indicate 5 iterations
   ;        to allow the network to relax into the correct state).
   ;        However, here we call it with a second argument of 1
   ;        in a loop so we can plot the network activation values
   ;        in between relaxation cycles:
   ;;

   (dotimes (call-net 5) ;; iterate Hopfield net 5 times
    (HopfieldNetRecall aHopfieldNetwork 1)  ;; calling with 1 iteration
    (plot-*inputCells*))

   ; delay a while for viewing the activation energy plots:
   (dotimes (delay 100000) nil))))))  ;; change delay for your system
```

Note that the example program is divided into two sections. The first section contains a reusable software module that can be used in larger application programs. The second section is throw away code to demonstrate the use of the Hopfield neural network module.

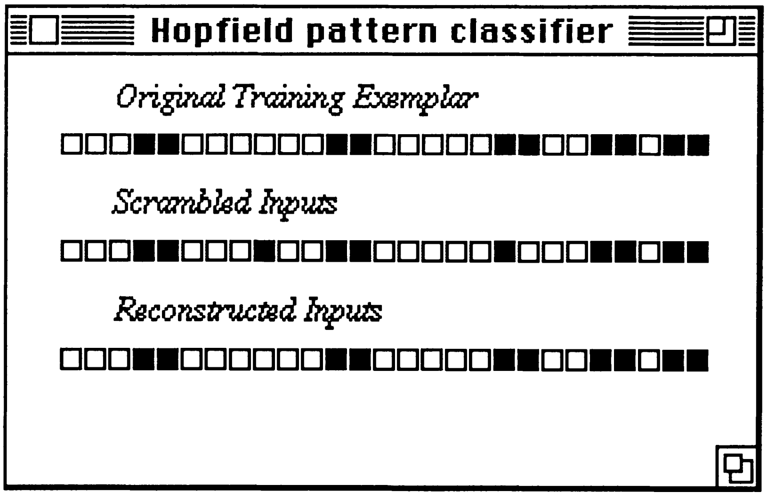
4.3 Executing the Hopfield Simulator

The program uses the generic plotting library listed in the second chapter. This plotting library will be used frequently in this book, so you may want to compile it and set it up to auto-load whenever you boot your LISP environment. After loading in the sample program listing in section 4.2, execute the function *test* to start the program.

The "throwaway" code at the bottom of the example program defines initial training patterns and forms the autocorrelation weight matrix to store these patterns in the network. The program then runs forever (until you break out of it) by cycling through the original training patterns creating new test patterns by randomly flipping bits in the input patterns. The data in the sample program is specified as values of zero or 1. The function *Hopfield-Init* converts these input values to the range of values from -1 to +1; this conversion is done automatically by the example program since it is easier to specify training data as values of zero or 1. While the program runs (see Figures 4.1, 4.2, and 4.3), you will see the original training exemplar as a row of data labeled Original Training Exemplar at the top of the graphics window, a row of scrambled data labeled Scrambled Inputs in the middle of the graphics window, and the recalled pattern labeled Reconstructed Inputs at the bottom of the graphics window. As the Hopfield network settles into a stable state, you will see the bottom pattern start to match up with the top reference pattern. Since the testing patterns are randomly generated, sometimes the generated pattern is so different from the reference pattern that the network settles in a stable state which is not quite equal to the original training reference pattern.

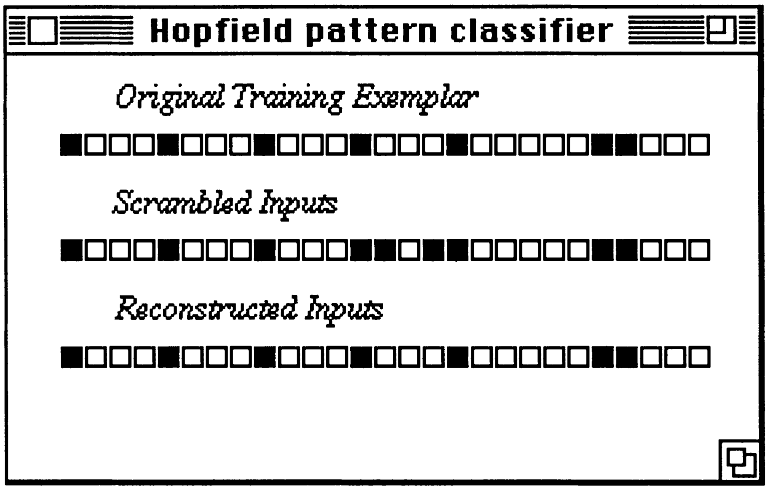
Figures 4.1, 4.2, and 4.3 show scrambled input patterns being reconstructed correctly into the original training exemplar pattern they most closely match. Figure 4.4 illustrates the failure to reconstruct the original image when the test input does not match any of the training exemplar patterns.

Familiarity with Hopfield neural networks will allow you to easily solve associative memory recall operations in your application programs. Simulated neural network components are relatively simple to modify when system requirements change. For example, if you build a face recognition security system for your company, new employees can be added to the system by adding a digitized image of their face to an image database and then recalculating the Hopfield network weight matrix used in the system. The weights that form the run-time behavior of neural networks can be treated as system data that can be recalculated off line to change the performance of neural network software modules.



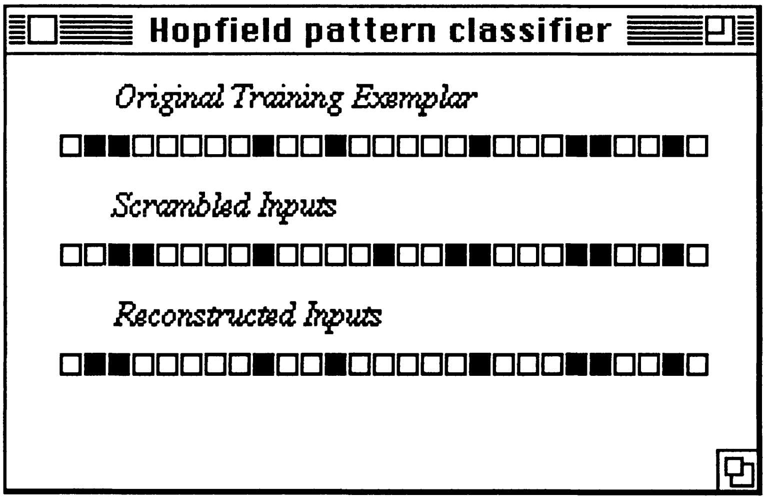
```
Original Training Exemplar
□□□■■□□□□□□■■□□□□□■■□□■■□■■
Scrambled Inputs
□□□■■□□□■□□■■□□□□□■□□□■■□■■
Reconstructed Inputs
□□□■■□□□□□□■■□□□□□■■□□■■□■■
```

Figure 4.1. Successful program execution with two pixels reversed. When the original training pattern is correctly recognized, the top and bottom rows of data should be identical.



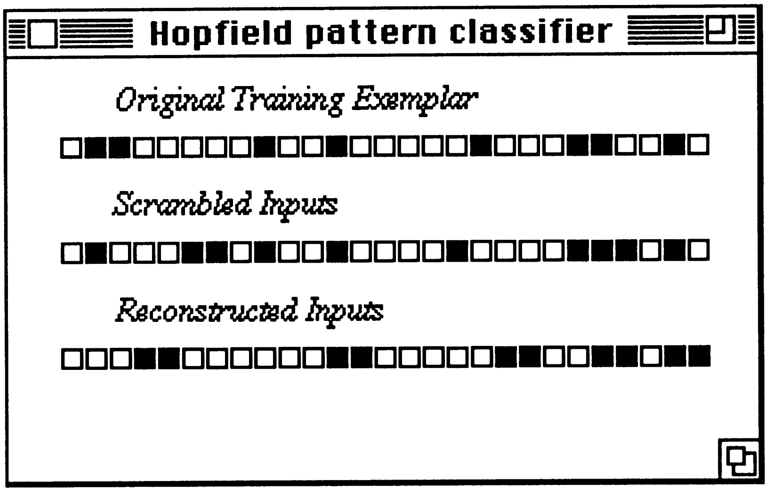
```
Hopfield pattern classifier

Original Training Exemplar
■□□□■□□□■□□□■□□□■□□□□□■■□□□
Scrambled Inputs
■□□□■□□□■□□□■■□■■□□□□□■■□□□
Reconstructed Inputs
■□□□■□□□■□□□■□□□■□□□□□■■□□□
```

Figure 4.2. Successful program execution with two pixels reversed.



```
Hopfield pattern classifier

Original Training Exemplar
□■■□□□□□■□□■□□□□□■□□□■■□□■□
Scrambled Inputs
□□■■□□□□■□□□□■□□■■□□□■■□□■□
Reconstructed Inputs
□■■□□□□□■□□■□□□□□■□□□■■□□■□
```

Figure 4.3. Successful program execution with five pixels reversed.



Original Training Exemplar

□■■□□□□□■□□■□□□□□■□□□■■□□■□

Scrambled Inputs

□■□□□■■□■□□■□□□□■□□□□■■■□■□

Reconstructed Inputs

□□□■■□□□□□□■■□□□□□■■□□■■□■■

Figure 4.4. Failed program execution with six pixels reversed. The reconstructed inputs do not match the original training pattern.

4.4 Projects

1. Use the Hopfield network simulator provided in this chapter to learn to recognize pictures. The pictures can be small digitized images. For example, you might have 10x10 pixel images represented by a 0 or a 1 for each pixel:

```
0001111000
0010000100
0100000010
1101101101
1000010001
1000000011
0101111010
0010001100
0001111000
0000000000
```

   The Hopfield network simulator assumes a one-dimensional input vector, so treat the 10x10 pixel two-dimensional picture array in this example as a single 100-pixel one dimensional array.
   If you have a high-performance computer (megabytes of memory with a fast floating point processor), you might try processing high-resolution digitized images. If you can capture digitized images and display them on your computer, write a program to convert the image data into a two-dimensional array of 0 and

1 pixel values and use this data to train a Hopfield neural network. Adjust the contrast and brightness of the images so that most of the pixel values are 0, and relatively few are 1.

2. Chapter 6 contains sample code for capturing the motion of the mouse cursor in a graphics window and converting that captured image into a list of screen coordinate points converting these points into a two-dimensional data array representing pixels that are black or white. After reading chapter 6 on handwriting recognition, combine the example program code provided therewith the Hopfield network software module provided in this chapter to build a system that uses Hopfield networks for recognition of hand-drawn characters. How do the training times between the two systems differ? Which system is the most accurate?

5 Speech Recognition Using Neural Networks

It is generally agreed that **natural language** will become the most common method for interacting with computers, increasing our reliance on natural language processing, the task of understanding written human language, and **speech recognition**, the task of converting human speech into written form. This chapter will focus on speech recognition - what it is, what speech features are used for recognition, its associated problems, and possible applications. We will also review a sample program for speech recognition, examining what it does, how it works, show sample data, create the training data for a neural network based speech recognition system, and consider ideas for expanding the system.

5.1 Problems Associated with Speech Recognition

Neural network speech recognizers suffer from the same lack of perfect understanding as human beings. In general, when we listen to speech, we do not understand every word. We use our knowledge of proper English syntax and grammar and our ability to guess what is forthcoming in a conversation to reduce the high **error rate** in recognizing speech by other people. The error rate is a measurement of the percentage of words that the neural network fails to recognize correctly. (One of my current research interests is the use of hybrid systems based on a neural network "front end" speech recognizer with "back end" natural language processing system to reduce the error rate of a speech recognition system).

Neural networks provide a new approach for solving the difficult problem of speech recognition. Unlike most general-purpose speech recognition systems which understand speaker-independent continuous speech, our system accepts as input digitized speech from a single speaker with clear breaks between the words. Our goal is to create a system that can be implemented easily using existing software modules using inexpensive computer hardware.

5.2 How Do Speech Recognition Systems Work?

Speech recognition systems extract **spectral** (frequency) information from speech data, and use these features to identify words provided in training examples. The sample program plot in Figure 5.1 shows raw speech data with the extracted frequency information appearing as black boxes at the bottom of the graphics window. This set of boxes represents a frequency range, with low frequency, or bass, on the left. The height of each individual bar indicates the amount of power in each part of the range. The scatterplot filling the graphics window shows the raw data when the two words *Yes* and *No* are spoken. The boxes within the graphics window enclose the raw data the sample program identifies as containing the actual words spoken; raw data outside the rectangles is discarded.

We will develop software to find the starting position of words, extract spectral features, and to use a delta rule neural network to classify the extracted spectral features as one of the words that the network has been trained to recognize. Several useful software techniques will be used in this chapter: the conversion of time domain data to frequency domain using **Fourier Spectral Analysis**, and **downsampling** data for low-resolution scatter plots to quickly view large amounts of data. Fourier Spectral Analysis extracts frequency (or spectral) information from time-series data. Either a Fast Fourier Transform (FFT) or a Direct Fourier Transform (DFT) can be used to extract frequency information from raw amplitude data. That information is displayed like the bass, mid-range, and high-frequency components of music that appear on a stereo mixer.

The example presented in this chapter can be duplicated with the use of an analog to digital converter to capture speech data. Typically these digitizers cost about $100 and work by accepting input from an amplified audio source such as the ear phone output from a small cassette tape recorder. The example system used a digitizer that captured about 6000 samples per second (6 KHz). A 6 KHz sampling rate allows resolution of sound to a frequency of 3 KHz (very low frequency

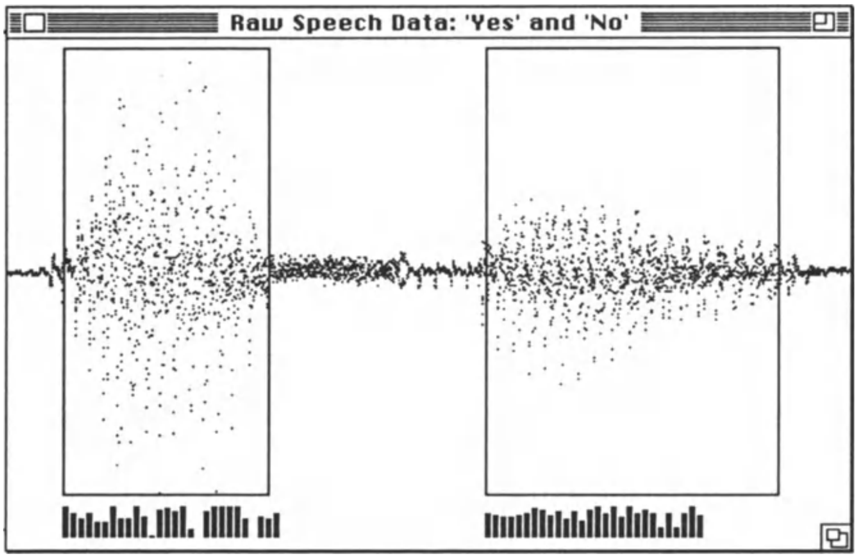


Figure 5.1. Scatter plot of raw speech data and spectral features from sample program. The spectral features (seen as vertical bars) calculated from the raw speech data will be input to the 'slab1' input neurons while training the neural network speech recognizer.

fidelity). Thus the example system uses only low frequency features of speech data. Raw data is processed by finding the word breaks. Each individual word is processed by calculating spectral features in a sampling window that is passed through the raw data for the word. The spectral features for each position of the sampling window are saved for input to the neural network.

The frequency information for each training example of a word requires storing only 28 integers which are scaled to the range 0.0 to 1.0 and used as input to a delta rule neural network. Figure 5.2 shows the delta rule simulator used with 28 input *slab1* neurons. The two output neurons correspond to the two words in the smaller two word training set. When the sample system was trained to recognize the numbers from zero to nine, the network had ten output neurons.

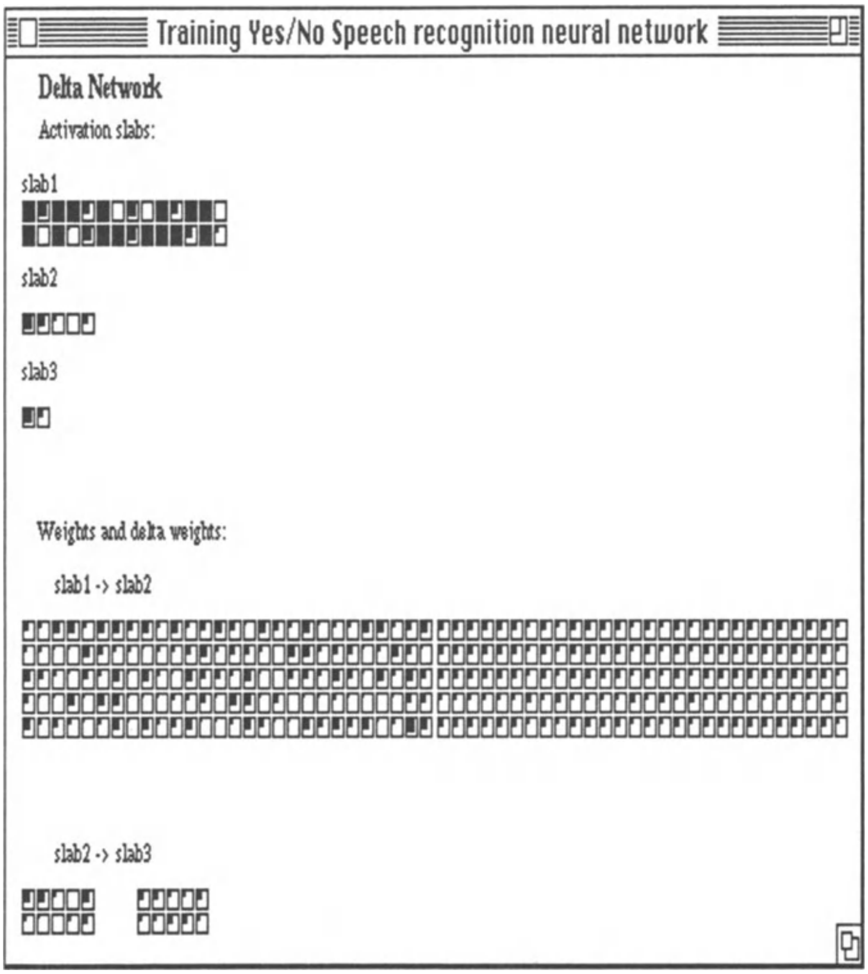

Training Yes/No Speech recognition neural network
Delta Network
Activation slabs:
slab1
slab2
slab3
Weights and delta weights:
slab1 -> slab2
slab2 -> slab3

Figure 5.2. Delta rule neural network simulation module developed in chapter 3 is being trained to recognize spectral features extracted from raw speech data.

The speech recognition software developed in this chapter has been used for two sets of trained words: the two words *yes* and *no* and the words for the range of single-digit numbers *zero* through *nine*. The simpler example is used in this chapter since it requires minimal computer resources (less than a half a megabyte of raw data for training examples and a training time of less than twenty minutes on a Mac II) and is reproducible by readers with minimal systems.

5.3 Example Program For Speech Recognition

The sample program listed in this section is very short since it uses the reusable software modules for plotting and delta rule neural network simulation developed in chapters 2 and 3.

```
;;
 ; Speech Recognition Software.  Requires Delta Rule module and Plotting Module
 ;
 ; Copyright 1990 by Mark Watson
 ;;

;;
 ; Utility for finding word breaks in raw speech data:  function word-breaks
 ; is called with an array of data values between 0 and 255.  Raw data values of
 ; 128 indicate that at the time the data value was sampled, there was no sound
 ; amplitude.  Sound amplitude values are averaged over 50 raw data values:
 ;;

(defun word-breaks (data)
  (defun off-mean (start-index)
    (let ((retVal 0))
      (dotimes (i 50)
        (setq retVal (+ retVal
                  (abs (- (aref data (+ start-index i)) 128)))))
      (/ retVal 50)))
  (let ((len (length data))
        (word-boundaries)
        start stop  (inside-word? nil))
    (dotimes (nn (truncate (/ (- len 50) 10)))
      (let ((n (* nn 10)))
        (if inside-word?
          (if (<= (off-mean n) 3)
            (setq inside-word? nil
                stop n
                word-boundaries (cons (list start stop) word-boundaries)))
          (if (>= (off-mean n) 11)
            (setq inside-word? t
                start (- n 50))))))
    (reverse word-boundaries)))
```

```
;;
; Plot a highly downsampled plot of raw data with word breaks: function
; raw-plot has two required arguments. The first is an array of raw speech
; data. The second argument is a list of word break boundaries. Raw data
; is plotted on a compressed horizontal axis producing a scatter plot:
;;

(defun raw-plot (data word-boundaries)
 (let* ((len (length data))
        (num-words (length word-boundaries))
        (plot-width 400)
        retVal ; used to return the spectral features
        (data-points-per-pixel (truncate (/ len plot-width))))
   (init-plot "Raw Speech Data: 'Yes' and 'No'" 430 292)
   (dotimes (i len)
     (let ((x (truncate (/ i data-points-per-pixel)))
           (y (+ (aref data i) 5)))
       (plot-line x y x y)))
   (dotimes (i num-words)
     (let ((start-of-word
            (truncate (/ (car (nth i word-boundaries))
                         data-points-per-pixel)))
           (end-of-word
            (truncate (/ (cadr (nth i word-boundaries))
                         data-points-per-pixel))))
       (let ((spectra
              (spectral-features
               data
               (car (nth i word-boundaries))
               (cadr (nth i word-boundaries)))))
         (setq retVal (cons spectra retVal))
         (plot-spectra spectra start-of-word 285))
       (plot-line start-of-word 5 start-of-word 260)
       (plot-line start-of-word 260 end-of-word 260)
       (plot-line end-of-word 260 end-of-word 5)
       (plot-line end-of-word 5 start-of-word 5)))
   retVal))

;;
; Process raw data: function spectra calls function word-breaks to calculate
; the starting position of each word in the raw times series data. Then
; function raw-plot is called to calculate the spectra and produce data plots:
;;

(defun spectra (data-set)
 (raw-plot data-set (word-breaks data-set)))
```

```
;;
; Extract spectral features: function spectral-features is called with an array
; of raw data values and the starting and ending array indexes for a word.
; A DFT is calculated to provide spectral energy levels (normalized
; between 0.0 and 1.0):
;;

(defun spectral-features (data-set start stop) ; use a DFT to calculate the
 (defun clamp (value)                          ; spectral features in desired
  (if (> value 0.5)                            ; frequency bins.
    (setq value 0.5)
    (if (< value -0.5)
      (setq value -0.5)))
  (+ value 0.5)) ; range of returned data: (0.0 1.0)
 (let* ((len (- stop start))
        (sample-size 256)
        (nSamples 4)
        (nBins 7)
        retVal temp tempList
        (x-offset (truncate (/ (- len sample-size) nSamples))))
  (dotimes (ns nSamples)
    (setq tempList nil)
    (dotimes (nb nBins)
      (let ((k (* nb 40)))
        (setq temp (complex 0 0))
        (dotimes (nn sample-size)  ; loop to calculate the DFT
          (let ((n (+ start (* nb 40) (truncate (/ (* ns sample-size) 2)))))
            (setq temp
                  (+ temp
                     (* (aref data-set n)
                        (complex
                         (cos (/ (* -6.28318 n k) sample-size))
                         (sin (/ (* -6.28318 n k) sample-size))))))))
        (setq temp
              (clamp
               (-
                (log
                 (+ (* (realpart temp) (realpart temp))
                    (* (imagpart temp) (imagpart temp))
                    0.0001))
                20.5))))
      (setq tempList (cons temp tempList)))
    (setq retVal (append tempList retVal)))
 (reverse retVal)))

;;
; Function plot-spectra plots a spectra at a given x y position:
;;

(defun plot-spectra (spectra x y)  ; plot vertical bar chart of spectral features at x,y
 (dolist (sf spectra)
  (let ((height (truncate (* 18 sf))))
   (plot-fill-rect x (- y height) 3 height 100))
  (setq x (+ x 4))))

;;
; Function train trains a 3 layer backprop network.
; This function has one optional argument; if this argument is not
; supplied, then the network is initialized. This function assumes the
; global variables spec1, spec2, spec3, spec4, spec5, and spec6 have been
; defined using the function spectra. The function DeltaLearn (defined
; in the neural network software module in chapter 3) is called in the autoplot
; mode so neural network state plots can be examined by holding down the
; mouse button during training.
;;

(defun train (&optional (restart 'yes))
 (init-plot "Training Yes/No Speech recognition neural network" 500 290)
 (if
  (equal restart 'yes)
  (setq temp (newdeltanetwork '(28 5 2))))
 (dotimes (ii 10000)
  (print (list "............." ii))
  (print
   (deltalearn
    temp
    (list
     (list (car spec1) '(1.0 0.0))
     (list (cadr spec1) '(0.0 1.0))
     (list (car spec2) '(1.0 0.0))
     (list (cadr spec2) '(0.0 1.0))
     (list (car spec3) '(1.0 0.0))
     (list (cadr spec3) '(0.0 1.0))
     (list (car spec4) '(1.0 0.0))
     (list (cadr spec4) '(0.0 1.0)))
    'yes)))) ; autoplot mode
```

5.4 Running the Speech Recognition System

There are two phases to running the example speech recognition system: training the system and testing it to recognize spoken words.

5.4.1 Procedure for Training the Example System

The training and testing data can be captured and copied to data files for later use to avoid repeating the time consuming process of capturing speech data and preprocessing the data to extract spectral features. The digitizer used in this example outputs unsigned 8-bit values (represented by integers ranging in value from 0 to 255). Raw data from an audio digitizer must be read into your LISP programming environment. One easy method for importing numeric data values (which will work in any LISP environment) is to use the software that controls your audio digitizer to dump out speech data as ASCII text with a space character used to separate the numbers. You can then edit this ASCII file adding the LISP code to bind the numbers in the file to a LISP variable. As an example, if you have the six numbers *120 105 202 137 240 99* in a file, you can edit the file to look like:

(setq aVariable '#(120 105 202 137 240 99))

Then loading this file will define the variable *aVariable* for use in your program. Another method for reading raw numeric data into your LISP environment would be to use the Common LISP macro *with-open-file.*

The following steps were used to prepare data for and train the example system:

1. Digitize six samples of the words *yes* and *no*. Set two examples aside for testing the trained neural network (testing a neural network with the original training data is meaningless). Copy each *yes/no* sample into a text file and read the data into your LISP environment setting the variables *data1*, *data2*, *data3*, *data4*, *data5*, and *data6* to the six raw training arrays.
2. Use the function *spectra* in the example program to calculate spectral features from the training data to use as training input to the neural network. You can do this by evaluating:

   ```
   (setq spec1 (spectra data1))
   (setq spec2 (spectra data2))
   (setq spec3 (spectra data3))
   (setq spec4 (spectra data4))
   (setq spec5 (spectra data5))
   (setq spec6 (spectra data6))
   ```

   in your LISP listener window. Note that the function *spectra* in the example program produces a downsampled plot of the raw speech input data as seen in the graphics window in Figure 5.1. In order to view the large number of raw data points in a single plot, every point in the sample data sets was plotted with the x-coordinate of the point downsampled: for example, if the the x-coordinates range in value from 1 to 1000, the x-coordinate can be divided by 10 to compress the plot by a factor of 10 in the x (horizontal) direction. The function *spectra* also returns a list of extracted spectral features so after evaluating these six statements, the LISP variables *spec1*, *spec2*, *spec3*, *spec4*, *spec5*, and *spec6* will be defined in your LISP environment. These spectral features will be used as the input training data for the speech recognition delta rule neural network.

3. Execute the function *train* in the sample program to train the speech recognition neural network. While the delta rule neural network simulator is executing, you can hold down the mouse button to get graphical snapshots showing the state of the network such as the one in Figure 5.2.

Creating the data and training this neural network is easy because of the small size of our recognized word set. As we increase the number of words to be recognized, not only do we require more training data, but **time-warping** of the input speech data starts to degrade the performance of the system. (Time warping refers to speaking slowly or quickly).

This example system works for a single speaker. It is fairly robust (given a small vocabulary) for continuous speech (running words together when we speak) because we calculate word breaks starting at a position in each word with a fairly high minimum volume (often losing interesting spectral features at the beginning of words).

With an audio digitizing board (which usually costs about $100 for either a Macintosh or an IBM-compatible), you can build simple speech recognition capabilities into your own software. It is left as an exercise to interface the software supplied with audio digitizers to your LISP environment.

5.4.2 Testing the Speech Recognition Program

The follow text was captured from the LISP listener window after executing the function *train* (bold text is user input):

```
? (deltaRecall temp (car spec1))
(0.9495446829902793 0.1470095102413525)
? (deltaRecall temp (cadr spec1))
(0.1366715347185966 0.9266094433725993)
? (deltaRecall temp (car spec5))
(0.912298760199873 0.1167470797627841)
? (deltaRecall temp (cadr spec5))
(0.2021959468513246 0.8044567381986862)
?
```

The LISP variable *spec1* contains the data for the first of four training data sets. The LISP variable *spec5* contains the data for the first testing data set. The function *deltaRecall* takes two arguments: a neural network object (as returned from the function *NewDeltaNetwork*) and a list of activation input values for the network. Function *deltaRecall* initializes the activation energy values of the input layer neurons and propagates these values through the network; a list of resulting output neuron activation energies is returned. Note that the neural network easily recognizes the words *yes* and *no* in the test data: we expect function *deltaRecall* to return a list with approximate values of (1.0, 0.0) when its input is a list of the spectral features of the word *yes* and a list with approximate values (0.0,1.0) when its input is a list of the spectral features of the word *no*.

5.5 Expanding the Sample Speech Recognition System

The example system can be expanded in two ways: allowing for time-warping of input data and increasing the number of training examples to decrease the system error rate.

The example system works when words are spoken at a constant rate. For example, if you train the system by speaking the words in the training data set quickly, the system will only recognize words spoken quickly. Research at the ATR laboratory in Japan and at Carnegie-Mellon University has shown that adding time delays to delta rule networks allows for temporal irregularities in speech input (Waibel, 1987).

The second improvement to the example system is to increase the amount of training data used. For a larger (perhaps 10 to 20 word) vocabulary system, a good training strategy would be to record a 100 samples of each word and record the frequency information to disk files for future use. Of the 100 examples of each word, set aside ten samples of each word for later testing of the system. Of the remaining 90 samples, use only two or three samples for each word to perform your initial delta rule neural network training. After the neural network has learned to recognize this small number of training cases, add a few more training cases from your set of 90 and retrain the network. Continue this process until your network can recognize all 90 training cases, then test it on the ten test cases.

5.6 Project

If you have access to an audio digitizer, first try repeating the experiment described in this chapter. Then choose an application which can be largely controlled by a small number of keyword commands and build a speech recognition module for this application.

6
Recognition Of Handwritten Characters

In this chapter, we will use a neural network pattern matcher to recognize handwritten characters. We will discuss required preprocessing of input data, present the sample program, show sample program output, and compare the neural network approach to a more conventional software approach to solving this problem.

The example program presented in this chapter illustrates the recognition of the handwritten numbers 0 through 9. As with the speech recognition system in chapter 5, the smaller the number of recognized symbols (or words), the easier it is to design and train a neural network discriminator.

There are many practical applications for the recognition of handwritten characters such as reading the dollar amount from checks or entering data without requiring the use of a keyboard. The goal of this chapter is to show how neural networks can easily solve a pattern-matching task which would otherwise be difficult using traditional software approaches.

6.1 Capturing Data and Preprocessing

The data captured for hand-drawn characters must be preprocessed for input into the delta rule neural network module which was developed in chapter 3. We use a mouse pointing device to capture data for handwritten characters in a graphics window; as a character is drawn while holding down the button on the mouse pointing device, the mouse's path in the graphics window is drawn in black. The resolution of this data collected from the graphic window is much higher than we want to use as input to a neural network, so we downsample the data for a single character onto a small grid.

Although a neural network could be trained to recognize characters that were not centered in the input grid, many more training examples would have to be provided to the network. Since the software required to center a character in a grid is simple, we will perform this alignment task as a data preprocessing step. The diagram in Figure 6.1 shows this initial data preprocessing performed in the example program.

The example program records the coordinates of points drawn with a mouse in the graphics window and downsamples these points, centers them in a small input grid, and provides them as input to a neural network.

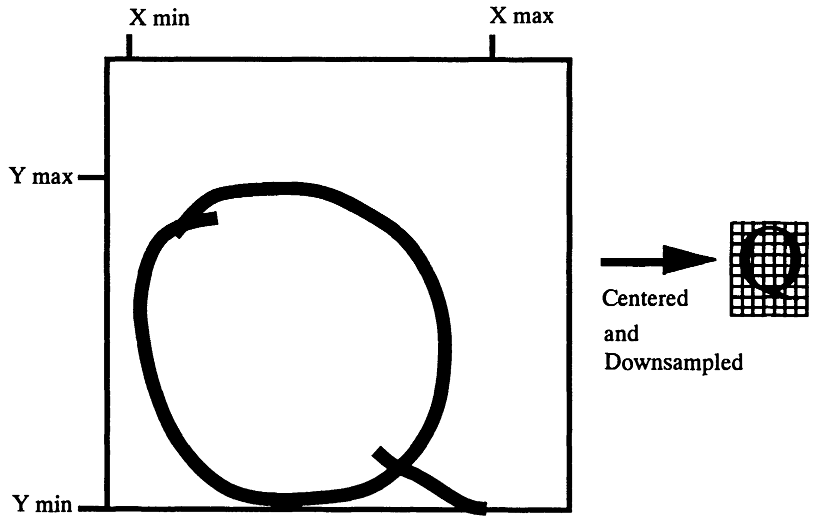


Figure 6.1. Preprocessing Hand-drawn Characters by centering and downsampling onto a uniform grid.

6.2 Handwriting Recognition Program

The example program is less than 100 lines of code since it uses the software modules developed in chapters 2 and 3 for graphics and neural network simulation. To train a neural network to recognize handwritten numbers, the input data can be organized as a two-dimensional array. If we consider our system input to be a two-dimensional field of 32x32 pixels, we can center the image of a hand-drawn number (which we will do by shifting and resizing the pattern) and then down sample (average or discard some data elements) the data to a smaller size, 6 by 6 pixels; each of the resulting 36 (= 6 * 6) pixels is represented by one of the 36 input neurons in our system. Furthermore, since we want our system to learn to recognize the ten digits 0 through 9, we will design our neural network to have ten neurons in the output slab. When we train the system with sample images of the number 1, we will force the activation level of neuron number one in the output slab to a high value and every other output slab neuron's activation will be set to zero. We will then allow the system to self-adapt its internal state to produce this desired output given the training image. This process is repeated for many different training examples.

The example program uses the global variable **symbols* to hold the print names for the symbols the network will be trained to recognize. The global variables *xScale* and *yScale* specify the size of the neural network input slab: the hand-drawn characters are centered and downsampled to a grid of dimensions *xScale* by *yScale* pixels. The function *start* sets up the data for a new delta rule neural network and saves the network description list in the global variable **H-network**. Function *start* also initializes a new graphics window (using the functions defined in chapter 2) and sets the global variable **training-list** to nil (setting the number of recorded training examples to zero).

The function *capturePoints* is used to capture the motion of the mouse cursor in the graphics window while the path of the mouse (while the mouse button is held down) is plotted in the graphics window. Function *capturePoints* discards points that are too close together and returns a list of x-y coordinate pairs specifying the points captured while the mouse button is held down.

The function *record* loops over all of the print names in the global variable list **symbol** prompting the user to draw the specified character in the graphics window by moving the mouse cursor with the mouse button held down to capture points. The global variable **training-list** is used to store the captured points of the entire set of symbols. The function *record* should be called many times to capture multiple training examples and store them on the list **training-list**.

The function *train* uses the complete delta rule neural network simulator listed at the end of chapter 3 to train the neural network stored in the variable **H-network**. Notice that both the learning rates and magnitude of random input noise are being adjusted to decreasing values after 100, 500, and 1500 training cycles through the training data stored in **training-list**.

The function *recall* is used to test the neural network after training is completed. This function first calls function *capturePoints* to collect the points for a hand-drawn character; then the function *DeltaRecall* (defined in the delta rule simulator) is used for pattern matching. If there is a good match to one of the training symbols, the appropriate training symbol print name is echoed in the Lisp listener window.

```
;;
 ; Capture hand-drawn numbers from 0 to 9 and train a neural network
 ; to recognize them.
 ;
 ;  Copyright 1990 by Mark Watson
;;

(setq *symbols* '("0" "1" "2" "3" "4" "5" "6" "7" "8" "9"))
(setq xScale 7)
(setq yScale 9)

(defun start ()
  (setq *H-network*
       (newdeltanetwork
        (list (* xScale yScale) 8 (length *symbols*))))
  (init-plot "Handwritten numbers")
  (setq *training-list* nil))

(defun capturePoints ()
  (defun differentPoints (p1 p2)
    (let ((xdif (- (car p1) (car p2)))
          (ydif (- (cadr p1) (cadr p2))))
      (> (+ (* xdif xdif) (* ydif ydif)) 5)))
 (clear-plot)
 (show-plot)
 (let ((xmin 9999) (ymin 99999) (xmax -9999) (ymax -9999)
       retList picList pos (oldPos '(-100 -100)))
   (dotimes (i 40000)
     (setq pos (plot-mouse-down))
     (if pos
       (if (differentPoints pos oldPos)
         (let ((x (car pos)) (y (cadr pos)))
           (plot-fill-rect (- x 2) (- y 2) 5 5 100)
           (if (< x xmin) (setq xmin x)
              (if (> x xmax) (setq xmax x)))
           (if (< y ymin) (setq ymin y)
              (if (> y ymax) (setq ymax y)))
           (setq retList
                 (cons pos retList)
                 oldPos pos)))))
   (if (< (abs (- xmin xmax)) 30) ;; special case for 1
     (setq xmin (- xmin 200)
          xmax (+ xmax 200)))
   (let ((xminB 4) (yminB 20)
         (xmaxB (+ 4 xScale)) (ymaxB (+ 20 yScale)))
     ;;
     ; Define a return picture list with the correct number of elements:
     ;;
     (dotimes (x xScale)
      (dotimes (y yScale)
        (setq picList (cons -0.4 picList))))
     (plot-line (- xminB 3) (- yminB 3) (+ xmaxB 3) (- yminB 3))
     (plot-line (+ xmaxB 3) (- yminB 3) (+ xmaxB 3) (+ ymaxB 3))
     (plot-line (+ xmaxB 3) (+ ymaxB 3) (- xminB 3) (+ ymaxB 3))
     (plot-line (- xminB 3) (+ ymaxB 3) (- xminB 3) (- yminB 3))
     (dolist (pos retList)
       (let ((x (car pos)) (y (cadr pos)))
         (let* ((xp (truncate (* 0.999 (* (/ (- x xmin) (- xmax xmin)) xScale))))
                (yp (truncate (* 0.999 (* (/ (- y ymin) (- ymax ymin)) yScale))))
                (index (+ xp (* yp xScale))))
           (setf (nth index picList) 0.4)
           (plot-line (+ 4 xp) (+ 20 yp) (+ 4 xp) (+ 20 yp))))))
   (dotimes (delay 200000) nil)   ;; delay a bit..
   picList))
;;
; Utility for recording training examples:
;;

(defun record ()
 (let (retVal (count 0) inputs)
   (dolist (sym *symbols*)
      (princ "Draw the character ")
      (princ sym)
      (princ " in the graphics window.")
      (terpri)
      (setq inputs (capturePoints))
      (setq *target-output* nil)
      (dotimes (ii (length *symbols*))
        (if (equal ii count)
          (setq *target-output* (cons  0.4 *target-output*))
          (setq *target-output* (cons -0.4 *target-output*))))
      (setq *target-output* (reverse *target-output*))
      (setq count (+ count 1))
      (setq retVal (cons (list inputs *target-output*) retVal)))
    (setq *training-list* (cons (reverse retVal) *training-list*))))

;;
; Utility for training the network using the complete delta rule
; module developed in chapter 3:
;;

(defun train (&optional (restart 'yes))
  (princ "Starting training session.  Hold down mouse for plots")
  (terpri)
  (setq defaultEidaList '(0.2 0.2 0.2 0.2))
  (setq defaultNoise 0.15)
  (if
    (equal restart 'yes)
    (setq temp (newdeltanetwork '(4 5 4))))
  (dotimes (ii 500000)
    (if (equal ii 100)
      (setq defaultEidaList '(0.1 0.1 0.1 0.1)
            defaultNoise 0.1))
    (if (equal ii 500)
      (setq defaultEidaList '(0.05 0.05 0.05 0.05)
            defaultNoise 0.08))
    (if (equal ii 1500)
      (setq defaultEidaList '(0.03 0.03 0.03 0.03)
            defaultNoise 0.03))
    (princ "RMS error = ")
    (princ
     (deltalearn
      *H-network*
      (car *training-list*)
      'yes)) ; autoplot mode
    (terpri)))

;;
; Utility for testing the network after training:
;;
```

```
(defun recall ()
  (let* ((pattern (capturePoints))
        (retVal (DeltaRecall *H-network* pattern)))
   (dotimes (i (length *symbols*))
    (if (> (nth i retVal) 0.1)
      (progn (princ "Symbol was ") (princ i) (terpri))))))
```

6.3 Training and Executing the Example Program

The hand writing recognition neural network uses both the generic plotting library and the complete delta rule simulator presented in chapters 2 and 3, so be sure to load these files before attempting to execute the sample program.

A training data base is created and tested by:

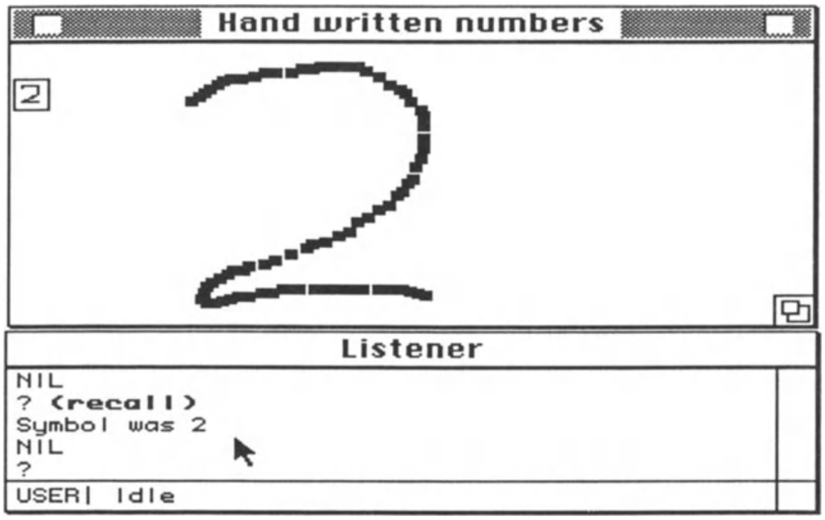
1. Execute the function *start* to initialize the neural network data structures and create a graphics window.
2. Execute the function *record* to define one complete set of training examples; repeat the execution of *record* several more times. Executing *record* saves samples of your hand writing for training the neural network.
3. Execute function *train* to train your network; the total error for each training cycle is printed after every pass through the training examples. Stop the execution of function *train* when this error becomes small and stops decreasing.
4. To test your neural network, execute the function *recall.*

The screen dump in Figure 6.2 shows the handwriting recognition neural network recalling stored patterns when shown new hand-drawn numbers. The preprocessing software allows the characters to be drawn in any position in the graphics window and in any size.

6.4 Alternate Problem-Solving Methodologies

This sample program is a good example of what current neural network technology does best: recognize (classify) patterns similar to patterns that the network has processed during a training cycle. It is interesting to compare and contrast the steps required for solving this example handwriting problem using both a neural network and a conventional software development approach. Neural networks provide a methodology for building pattern classifiers quickly without specialized programming. Given the problem of writing a *conventional* computer program for recognizing the characters for the numbers 0 through 9, we might take the following steps:

1. Write the software to track mouse movements on your computer system and store the path that the mouse followed.



```
NIL
? (recall)
Symbol was 2
NIL
?
USER| Idle
```

Figure 6.2. Screen dump from the example program showing a hand-drawn character in the top window and the sample program output in the bottom window correctly classifying the character.

2. Write the simple functions to center the hand-drawn character and down sample it onto a smaller grid.
3. Display many example down sampled characters and decide which features most clearly differentiate the different characters.
4. Use rules such as: *0 and 8 are vertically symmetric, if the height of the character is much greater than the width the character is a 1, if no pixels are recorded in the lower right part of the down sampled grid, the character is probably a 7*, etc.
5. Prototype/refine the system.
6. Test the system performance.
7. Repeat steps 3, 4, 5, and 6 as many times as necessary.

Note that we had to perform steps 1 and 2 in our neural network example, but then we used available software packages (in this case examples developed earlier in this book) to engineer a good recognition system without any further programming effort.

Remember that trained neural network systems can be easily embedded in larger LISP programs either by using the *ReadDeltaNetwork* function presented in chapter 3, storing trained networks as disk files and loading them into memory as needed, or by embedding them directly in LISP (or any other language) by allocating storage for the connection weights directly in the source code and then initializing the values for the weights as constants in the program. I have embedded neural networks in C programs by writing the network weights out to a disk file in the same format used to initialize static arrays, then editing the neural network (as represented by the connection weights) into an application program as statically initialized data. A run-time delta network interpreter requires less than fifty lines of code to process the network inputs.

A further advantage of building recognition systems with neural networks is that if the problem domain changes (e.g., the letters *A* through *F* need to be scanned from a consumer information questionnaire, or the dollar sign ($) and period (.) are needed for reading the dollar amounts from handwritten checks), the system can be retrained by simply recording more training data and repeating the training process.

Even though the training time can be considerable, recall time (to recognize patterns after the training has been finished) is fast: one forward pass through the delta rule network. To calculate this response time of a trained network, write down the size of each slab of neurons in the network and multiply these numbers together: the result will be the required number of multiplication and addition operations. The Sigmoid function must also be evaluated once for every neuron in the network; using a table lookup will speed up the Sigmoid calculations significantly (if you use tables for the Sigmoid and derivative of Sigmoid functions, each table should have at least 2000 elements). If you know how long it takes for these operations on your computer, you can get a good estimate of the time required for pattern recognition. Often preprocessing steps (such as extracting spectral features from speech data) are also computationally expensive.

6.5 Project

Build a system that can read handwritten numbers from checks. You will need to write preprocessing code to figure out where the breaks are between characters and pass each character to a neural network character classifier. Try to catch errors dealing with illegal entries. This is a difficult project because of the requirement of isolating the individual characters before processing with a neural network.

7
Adaptive Neural Networks

The neural networks studied in previous chapters have had a simple flow of information from input to hidden to output neurons through connections without loops and feedback. They were trained by **supervised learning** techniques where the desired system output was provided during training. Another interesting type of neural network performs **unsupervised learning** where the system is shown inputs with no desired outputs. The system searches for similar features in the training inputs to group them into categories where members of a single category share common features. These networks are called **adaptive resonance theory** (ART) networks (Carpenter and Grossberg, 1987). Carpenter and Grossberg have extended their theory to process inputs with a dynamic range. The sample program listed in this chapter uses this newer **gray-scale** Adaptive Resonance Theory (ART2) which allows inputs to have any value between 0.0 and 1.0 (thus the name *gray-scale*). This sample program uses the variable naming conventions originated by Gail Carpenter and Stephen Grossberg in their paper introducing the theory for ART2 (Carpenter and Grossberg, 1987). The real time dynamic behavior of ART2 networks is fascinating to watch; they exhibit interesting behavior when they have few output neurons and are shown a large number of input patterns. The network will reshuffle existing categorizations when necessary.

7.1 Theory

When an ART2 neural network remembers an input pattern in **long-term memory** (i.e., the weights in the network, not the neuron activation values), it stores the pattern in the weights. When several patterns are classified as belonging to the same category, the weights store an average of the input patterns in that category. An ART2 model creates new categories when a new input pattern is initially classified as belonging to an existing category but there is a **mismatch** between the recalled average pattern for the existing category and the new input pattern. The network disallows the current categorization (i.e., the network classification is **reset**) and forces a different output category.

In ART2 networks, each possible output category used for classifications is represented by one output neuron in the network. Learning and recall are the same operation in an ART2 network: long-term memory is slightly modified when classifying a new input pattern into an existing category. A rough analogy is our ability to recognize a building in a foreign country as being a house; our categorization of a new structure as a house slightly changes our perception of what houses look like.

7.2 ART2 Dynamics

When an ART2 neural network categorizes an input pattern, a generic pattern is recalled for that category which is a rough average of the input patterns forming the category. Figure 7.1 shows the input pattern (0.0, 0.3, 0.9, 0.3, 0.0) being placed in category 2 with no mismatch between it and existing patterns in category 2. If there is a significant mismatch (the allowed magnitude of this mismatch is controlled by the *vigilance* parameter) between the input pattern being categorized and the recalled pattern, a *reset* condition occurs. The screen image in Figure 7.2 shows an example of an input pattern not being accepted into either category 1 or category 3. The example program first started to classify the pattern as being in category 3, but there was too large a mismatch between the generic recalled category 3 pattern and the new input pattern being tested. Similarly, an attempt to classify the new input pattern as a category 1 pattern triggered a reset condition. The pattern is accepted into category 2 although the match is not perfect (notice that the reset indicator is nonzero) signifying that the new pattern differs slightly from previous patterns categorized as belonging to category 2.

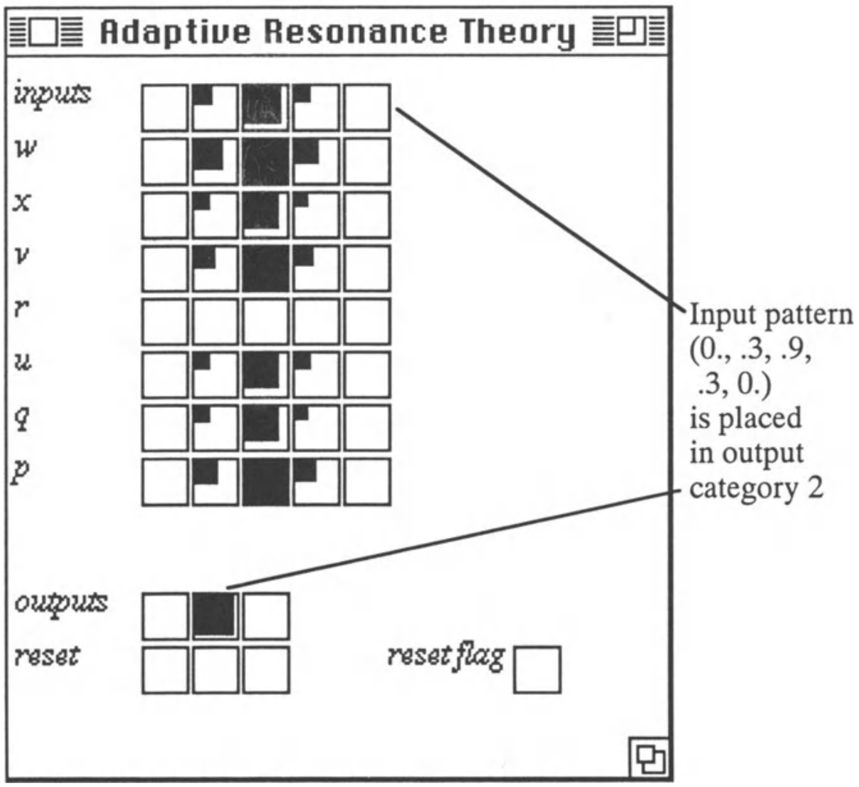

Adaptive Resonance Theory
inputs
w
x
v
r
u
q
p
outputs
reset
reset flag
Input pattern (0., .3, .9, .3, 0.) is placed in output category 2

Figure 7.1. Example ART2 Screen Image. The ART2 model selects category 2 for an input pattern.

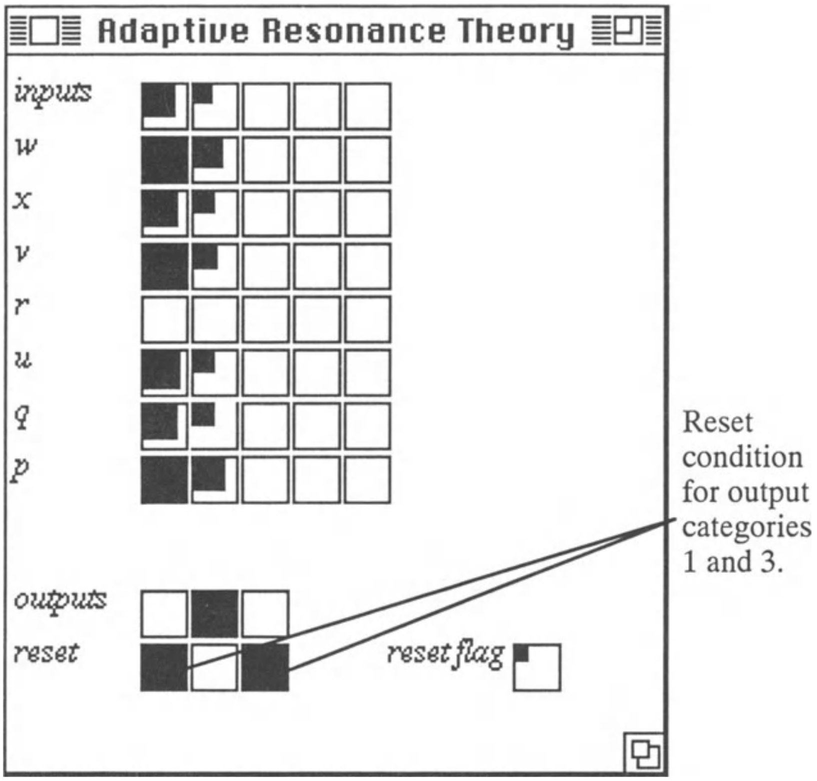

Adaptive Resonance Theory
inputs
w
x
v
r
u
q
p
outputs
reset
reset flag
Reset condition for output categories 1 and 3.

Figure 7.2. ART2 Selection of Category 2. Initial classification as categories 1 and 2 resulted in a mismatch between input pattern and stored memory of categories 1 and 3.

ART2 networks are fundamentally different from feed forward delta rule networks. Certainly the action of delta rule networks is more predictable since the network training is supervised by supplying both system inputs and outputs. ART2 neural networks self-organize their weights internally to define a classification scheme for patterns which they process in real time. Learning and recall occur simultaneously whenever a pattern is input to the network. Patterns which closely match an existing category do not disturb the internal state of the network very much. A pattern very dissimilar to any seen before has a major impact on both the internal state of the network and its functional properties that will be manifested in the future: a new pattern dissimilar to previous ones will often initially be classified as being similar to an old pattern. The classification of a pattern causes an associative memory recall (using the output category code as a key) of a rough average of all previous inputs which have been placed in this category.

Figure 7.3 shows the input pattern (0.0, 0.25, 0.9, 0.1, 0.0) being categorized as belonging to category 2. There is a slight mismatch to the input pattern (0.0, 0.3, 0.9, 0.3, 0.0) seen in Figure 1, but both patterns are placed in the same category.

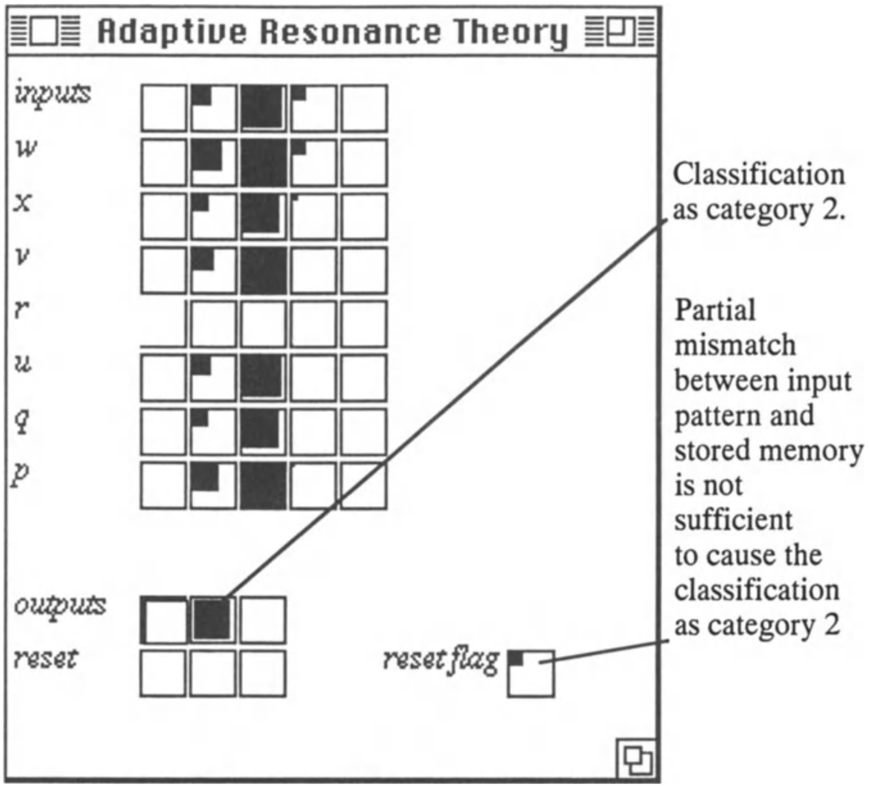

Adaptive Resonance Theory
inputs
w
x
v
r
u
q
p
outputs
reset
reset flag
Classification as category 2.
Partial mismatch between input pattern and stored memory is not sufficient to cause the classification as category 2

Figure 7.3. Selection of Category 2 for Input Pattern. Partial mismatch between the input pattern and the stored memory image for category 2 is indicated by the nonzero reset neuron in the lower right corner of window.

7.3 ART2 Model Equations

Figure 7.4 shows the Carpenter and Grossberg ART2 model equations used later in the sample ART2 program. The model constants (*a*, *b*, *c*, *d*, *e*, *f*, *theta*, *alpha*, and *vigilance*) should not in general be modified. The one-dimensional arrays *p*, *q*, *u*, *v*, *w*, and *x* are used to calculate the **short-term memory** (i.e., the activation energy) in the output neurons (stored in array *y*). The one-dimensional array *I* holds an input pattern. The two dimensional arrays wDown and wUp are used to store the weights in the network which are the network's long-term memory. The function *L2NORM* is calculated as the sum of the squares of the elements of an array plus a small value to prevent divide by zero error conditions if the array elements are all zero.

$$p_i = u_i + \sum_j g(y_j) * wDown_{ji}$$

$$q_i = \frac{p_i}{L2NORM\ (p)}$$

$$u_i = \frac{v_i}{L2NORM\ (v)}$$

$$v_i = f(x_i) + b * f(q_i)$$

$$w_i = I_i + a * u_i$$

$$x_i = \frac{w_i}{L2NORM\ (w)}$$

$$f(x) = \begin{cases} 0 & \text{if } 0 \leq x < theta \\ x & \text{if } x \geq theta \end{cases}$$

$$y_i = \sum_j p_i * wUp_{ij}$$

$$wDown_{ji} = \sum_i \sum_j learning\_rate * d * p_i * wDown_{ji}$$

$$wUp_{ij} = \sum_i \sum_j learning\_rate * d * p_i wUp_{ij}$$

Figure 7.4. Carpenter and Grossberg ART2 Model Equations.

7.4 ART2 Program Listing

The following listing shows the example program which produced the screen images used in this chapter (some of the comments refer to citations in Carpenter and Grossberg, 1987). The example program uses several global variables to store model constants, the network size (*nInputs* and *nOutputs*), and learning rates (*upLR* and *downLR*). The use of global variables makes the debugging of complex mathematical models like ART2 easier in an interactive environment, but is in general not recommended. Any bad effects of the use of global variables are mitigated by placing the ART2 model program in it's own Common LISP package. In the example program, only the functions*ART2-init*,*ART2*, and*ART2-postprocess* are exported to the default Common LISP package *user*.

This program uses the variable naming conventions used by Carpenter and Grossberg in their published papers on ART and ART2. They often use single-letter variable names. The reader who is interested in adaptive neural networks will find it easier to understand and modify the example program because of the use of the same naming conventions used in the ART2 literature.

The function *ART2-init* accepts two arguments: the number of input neurons and the number of output neurons. This function allocates floating point array storage for the model.

The function *frandom* has two arguments: a lower bound and an upper bound between which a random floating point number is generated and returned as the value of the function.

The function *findLargestOutput* examines each output neuron and returns the index of the output neuron with the largest activation energy (as defined in chapter 3). The ART2 model uses the array *reset* as a flag array for indicating which output neurons are illegal (because of a reset condition). This function ignores output neurons which have their reset flag set to true.

The function *g* is called with one argument (the index of an output neuron) and returns as its value the model parameter *d* if the indexed output neuron has the largest value, has not been reset, and has an activation energy greater than the model constant *resetThreshold*. Otherwise, this function returns 0.0 as its value.

The function *sigmoid* in the sample program is not the same sigmoid function that we saw in chapter 3. This function returns the value of zero if its argument is less than the model cutoff value of *theta*, otherwise it returns its input argument as the return value for this function.

The function *L2NORM* has as input, a vector, and the length of the vector. This function calculates the sum of the square of each element in the input vector and returns the square root of this sum as the functions return value.

The function *F1STMcycle* updates the F1 short-term memory arrays *p*, *q*, *u*, *v*, w, *x*, and *r*.

The function *F2STMcycle* updates the short-term memory array y which is the output of the network indicating the classification of the input pattern.

The function *updateWeights* updates the model weight arrays: *wDown* and *wUp*.

The function *competitiveF2* finds the index of the output neuron with the largest activation energy (using function *findLargestOutput*) and if the value of the array *y* for this index is greater than the model constant *resetThreshold*, every other element of the array *y* is set to zero.

The function *OneCycle* executes one complete learning/recall cycle.

The function *testReset* checks for an F2 short-term memory reset condition caused by too large of a mismatch between an input pattern and the recalled pattern triggered by this input. If a reset condition exists, the index of the output neuron with the largest activation energy is used to set the corresponding element of the *reset* array to a value of *true*.

The function *zeroActivations* zeros out all short-term memory arrays in the ART2 model.

The function *setPattern* has a single input argument: a list containing *nInputs* floating point numbers with values between 0.0 and 1.0. The function *zeroActivations* is called to zero out all short-term memory arrays and the input pattern is copied into the model array *p* which is used to hold the input activation values.

The function *initNetwork* zeros all short-term memory arrays and resets long-term memory (i.e., the weight arrays *wUp* and *wDown*) to small random numbers. Calling this function completely resets the simulated ART2 network.

The function *learnPatterns* uses a set of input patterns stored in the list *testPatterns* to train the network. The variable *testPatterns* is defined before the definition for function *learnPatterns* in the sample program. The main program ART2 calls this function, so new sample inputs can be used by changing the definition of *testPatterns*.

The function *plotActivations* is a utility function for plotting the activation values stored in a one-dimensional array with a title at a specified location in the graphics window. This function uses the graphics utilities developed in chapter 2.

The function *updateScreen* calls function *plotActivations* for every short-term memory model array to get the plots shown in Figures 7.1, 7.2, and 7.3.

The function *ART2* is the main test function in the sample program. It is called with no arguments and cycles through the training patterns stored in the variable *testPatterns* 50 times.

The function *ART2-Postprocess* produces the output summary plots.

```
;;
;      Adaptive Resonance 2 Model in Common Lisp.
;      Copyright 1990 by Mark Watson.
;
;      Based on the Adaptive Resonance Theory
;      by Gail Carpenter and Stephen Grossberg.
;;


;;
; Define a separate name space package for this software module:
;;
(make-package :art2)
(export '(ART2 ART2-postprocess ART2-init)  :user)

(defun ART2-init (numInputs numOutputs &optional trainingPatterns)
 ; Check for specified training patterns:
 (if trainingPatterns
   ; Make sure the number of input neurons agrees with
   ; the size of the training patterns:
   (if (equal (length (car trainingPatterns)) numInputs)
     (setq testPatterns trainingPatterns)
     (print
      (list
       "ERROR: bad input to ART2-init. numInputs should have been"
       (length (car trainingPatterns)))))
   ; No specified training patterns: use the default set
   ; defined in this package:
   (if (not (equal (length (car testPatterns)) numInputs))
     (print
      (list
       "ERROR: bad input to ART2-init. numInputs should have been"
       (length (car testPatterns))))
     ; Specified number of input neurons agrees with
     ; the size of the default training patterns defined
     ; in this package; proceed with defining network data:
     (progn
       ; Define the network size:
       (setq nInputs numInputs)
       (setq nOutputs numOutputs)
       ; Array storage allocation:
       (setq input (make-array (list nInputs)))
       (setq w (make-array (list nInputs)))
       (setq x (make-array (list nInputs)))
       (setq v (make-array (list nInputs)))
       (setq r (make-array (list nInputs)))
       (setq u (make-array (list nInputs)))
       (setq q (make-array (list nInputs)))
       (setq p (make-array (list nInputs)))
       (setq temp (make-array (list nInputs)))
       (setq resetVal (make-array (list 1)))
       (setq y (make-array (list nOutputs)))
       (setq reset (make-array (list nOutputs)))
       (setq resetCounter (make-array (list nOutputs)))
       (setq nCategories (make-array (list nOutputs)))
       (setq temp2 (make-array (list nOutputs)))
       (setq wUp (make-array (list nInputs nOutputs)))
       (setq wDown (make-array (list nOutputs nInputs)))
       ; Global variable to remember input patterns and
       ; their associated output category code for plotting
       ; by function ART2-Postprocess:
       (setq *learned-categories* nil)))))

; Neuron size for plots:

(setq PEsize 18)
(setq PEsizem1 (- PEsize 1))
(setq PEsizep1 (+ PEsize 1))

; Model constants:

(setq a 0.5)
(setq b 0.2)
(setq c -1.0)
(setq d 0.4)
```

```
(setq e 0.04)
(setq theta 0.3) ;; 1 / (sqrt(nInputs))
(setq vigilance 0.94)
(setq alpha 1.0)

(setq resetThreshold 0.05)

(setq upLR 0.12)
(setq downLR 0.12)

 ; Floating point  random numbers:

(defun frandom (low high)
  (let ((range (- high low)))
    (+ (* (/ (random 1000) 1000.0) range) low)))

(defun findLargestOutput (&aux (maxIndex 0) (mVal (aref y 0)))
  (dotimes (j nOutputs)
    (if (and
         (> (aref y j) mVal)
         (not (aref reset j)))
      (setq mVal (aref y j)
            maxIndex j)))
  maxIndex)

; The following function returns d if (aref y index) is the
; largest value in array y AND (aref y index) has not been reset:

(defun g (index &aux j mVal (maxIndex (findLargestOutput)))
  (if (and
       (equal index maxIndex)
       (not (aref reset maxIndex))
       (> (aref y maxIndex) resetThreshold))
    d
    0.0))

; Threshold function:

(defun sigmoid (x)
  (if (> x theta)
    x
    0.0))

; L2 Norm of a vector:
```

```
(defun L2NORM (v vLength &aux (sum 0.0) i j)
  (dotimes (i vLength)
    (setq sum (+ sum (* (aref v i) (aref v i)))))
  (+ (sqrt sum) 0.001))

; Update F1 STM arrays:

(defun F1STMcycle (&aux i j sum norm max1 max2)
  ; Calculate p from u input and backwards feed back:
  (dotimes (i nInputs)
    (setq sum 0.0)
    (dotimes (j nOutputs)
      (setq sum (+ sum (* (g j) (aref wDown j i)))))
    (setf (aref p i) (+ (aref u i) sum)))
  ; Update q using eq. 5
  (setq norm (+ (L2NORM p nInputs) e))
  (dotimes (i nInputs)
    (setf (aref q i) (/ (aref p i) norm)))
  ; Update u using eq. 6:
  (setq norm (L2NORM v nInputs))
  (dotimes (i nInputs)
    (setf (aref u i) (/ (aref v i) norm)))
  ; Update v using eq. 7:
  (dotimes (i nInputs)
    (setf (aref v i) (sigmoid (+ (aref x i) (* b (sigmoid (aref q i)))))))
  ; Update w using eq. 8:
  (dotimes (i nInputs)
    (setf (aref w i) (* alpha (+ (aref input i) (* a (sigmoid (aref u i)))))))
  ; Update x using eq. 9:
  (setq norm (+ (L2NORM w nInputs) e))
  (dotimes (i nInputs)
    (setf (aref x i) (/ (aref w i) norm)))
  ; Calculate reset r from eq. 20:
  (setq max1 -1000.0 max2 -1000.0)
  (dotimes (j nInputs)
    (progn
      (if (< max1 (aref u j)) (setq max1 (aref u j)))
      (if (< max2 (aref p j)) (setq max2 (aref p j)))))
  (setq max1 (+ max1 0.001))
  (setq max2 (+ max2 0.001))
  (dotimes (i nInputs)
    (setf
     (aref r i)
     (- (/ (aref u i) max1) (/ (aref p i) max2))))
```

```
; Update F2 STM storage:

(defun F2STMcycle (&aux i j sum)
  (dotimes (j nOutputs)
    (progn
      (setq sum 0.0)
      (dotimes (i nInputs)
        (setq sum (+ sum (* (aref p i) (aref wUp i j)))))
      (setf (aref y j) sum)
      (if (aref reset j) (setf (aref y j) -0.1)))))

; Update weights:

(defun updateWeights (&aux i (j (findLargestOutput)))
  (if (> (g j) 0.02)
    (dotimes (i nInputs)
      (setf
       (aref wDown j i)
       (+ (aref wDown j i)
          (*
           downLR
           d
           (- (aref p i) (aref wDown j i)))))
      (setf
       (aref wUp i j)
       (+
        (aref wUp i j)
        (*
         upLR
         d
         (- (aref p i) (aref wUp i j))))))))

; Competitive learning at slab F2:

(defun competitiveF2 (&aux i (jj (findLargestOutput)))
  (if (> (aref y jj) resetThreshold)
    (dotimes (i nOutputs)
      (if (not (equal i jj))
        (setf (aref y i) 0.0)))))

; Run one full cycle:

(defun OneCycle (&aux i j)
  (F1STMcycle)
  (testReset)
  (competitiveF2)
  (F2STMcycle)
  (updateWeights)
  (competitiveF2)
  (updateScreen))

; Check for an F2 reset condition:

(setq skipReset nil)

(defun testReset (&aux (res 0.0) (norm1 0.0) (norm2 0.0)
                       (n1 (+ (L2NORM p nInputs) e)) n2 temp)
  (if (and
       (> n1 0.2)
       (not skipReset))
    (if (> learningCycleCounter 1)
      (if (> (aref y (findLargestOutput)) 0.25)
        (setq res (* 3.0 (L2NORM r nInputs))))  ; was 3.0
      (setq skipReset nil)))
  (setf (aref resetVal 0) res)
  (plotActivations "reset flag" 190 220 resetVal 0.0 1.0)
  (if (> res 1.0)
    (progn
      (print (list "Vigilance reset =" res "  Learning cycle ="
                   learningCycleCounter))
      (setq maxIndex (findLargestOutput))
      (setf (aref reset maxIndex) 1)
      (setf (aref resetCounter maxIndex) 80))
    (dotimes (i nOutputs)
      (setf (aref resetCounter i) (- (aref resetCounter i) 1))
      (if (< (aref resetCounter i) 0)
        (progn
          (if (aref reset i)  (setq skipReset t))
          (setf (aref reset i) nil)))))
  (setq skipReset nil)) ;; temporary

; Zero activations:

(defun zeroActivations ()
  (dotimes (i nInputs)
    (setf (aref w i) 0.0)
    (setf (aref x i) 0.0)
    (setf (aref v i) 0.0)
    (setf (aref r i) 0.0)
    (setf (aref u i) 0.0)
    (setf (aref q i) 0.0)
    (setf (aref p i) 0.0))
  (dotimes (i nOutputs)
    (setf (aref y i) 0)
    (setf (aref reset i) 0)
    (setf (aref resetCounter i) 0)))

; Set up a learning pattern in the input neurons:

(defun setPattern (pl &aux (len (length pl)))
  (if (not (equal len nInputs))
    (print (list "Error in setPattern input:" pl))
    (progn
      (setq learningCycleCounter 0)
      (zeroActivations)
      (dotimes (i len)
        (setf (aref input i) (+ (pop pl) (frandom -0.08 0.08)))))))

; Initialize the network:

(defun initNetwork ()
  (zeroActivations)
  (dotimes (j nOutputs)
    (progn
      (dotimes (i nInputs)
        (setf
         (aref wUp i j) (frandom 0.05 0.1)
         (aref wDown j i) (frandom 0.01 0.03)))
      (setf (aref nCategories j) 0))))

; Define some test patterns:

(setq testPatterns
      '(
        (0.0 0.4 0.9 0.4 0.0)
        (1.0 0.3 0.0 0.3 1.0)
        (0.75 0.5 0.02 0.05 0.07)
        (0.02 0.41 0.91 0.36 0.08)
        (1.1 0.12 0.0 0.3 1.0)
        (0.0 0.2 0.9 0.4 0.0)
        (0.93 0.31 0.0 0.78 1.0)
        (0.6 0.6 0.02 0.05 0.07)
        (0.79 0.54 0.02 0.08 0.07)
        (0.0 0.45 0.7 0.25 0.0)
        (1.0 0.3 0.0 0.3 0.65)
    (0.0 0.4 0.9 0.4 0.0)
    (0.0 0.5 0.75 0.5 0.0)
    (1.0 0.2 0.0 0.4 0.8)
    (0.82 0.46 0.02 0.09 0.01)))

; Cycle through all training patterns once:

(defun learnPatterns (num)
  (dolist (p testPatterns)
    (print (list "Inputs:" p))
    (setPattern p)
    (dotimes (i num)
      (setq learningCycleCounter (1+ learningCycleCounter))
      (OneCycle)
      (updateScreen))
    (setq *learned-categories*
         (cons (list p (findLargestOutput))
               *learned-categories*))))

 ; Graphics support:

(defun plotActivations (title x y data dmin dmax)
  (let ((size (array-dimension data 0)) (ypos 0) (xpos x))
    (plot-string-italic (- x 47) (+ y 7) title)
    (dotimes (i size)
      (if (< size 20)
        (setq ypos y xpos (+ x (* i PEsizep1)))
        (if (< i (/ size 2))
          (setq ypos (- y 7) xpos (+ x (* i PEsizep1)))
          (setq ypos (+ y 2)  xpos (+ x (* (- i (/ size 2)) PEsizep1)))))
      (plot-size-rect
       xpos ypos PEsizem1 PEsizem1
       (max
        1
        (truncate (* (/  (- (aref data i) dmin) (- dmax dmin)) PEsize))))
      (plot-frame-rect xpos ypos PEsize PEsize))))

(defun updateScreen ()
  (plotActivations "inputs" 50 10 input 0.0 1.0)
  (plotActivations "w" 50 30 w 0.0 1.0)
  (plotActivations "x" 50 50 x 0.0 1.0)
  (plotActivations "v" 50 70 v 0.0 1.0)
  (plotActivations "r" 50 90 r 0.0 1.0)
  (plotActivations "u" 50 110 u 0.0 1.0)
  (plotActivations "q" 50 130 q 0.0 1.0)
 (plotActivations "p" 50 150 p 0.0 1.0)
 (plotActivations "outputs" 50 200 y 0.0 1.0)
 (dotimes (i nOutputs)
  (if (aref reset i)
   (setf (aref temp2 i) 1.0)
   (setf (aref temp2 i) 0.0)))
 (plotActivations "reset" 50 220 temp2 0.0 1.0))

; Main test program:

(defun ART2 ()
 (setq *learned-categories* nil)
 (init-plot "Adaptive Resonance Theory" 250 290)
 (initNetwork)
 (learnPatterns 50))

;;
;      Standalone program to plot the input
;      patterns and associated output category
;      codes produced by executing program ART2:
;;

(defun ART2-Postprocess (&aux (width 130) category-counts)

 (defun plot-input (x-org y-org values)
  (let ((delta-x (/ (- width 20) (length values))))
   (plot-line x-org y-org x-org (- y-org 20))
   (plot-line x-org y-org (+ x-org (- width 40)) y-org)
   (dotimes (n (- (length values) 1))
    (plot-line
     (+ x-org (* n delta-x))
     (- y-org (* 20 (nth n values)))
     (+ x-org (* (1+ n) delta-x))
     (- y-org (* 20 (nth (1+ n) values)))))))

 (setq category-counts (copy-tree '(0 0 0)))
 (init-plot "ART2: Category codes for input patterns" 382 270)
 (plot-string-italic 20 15 "Category 1")
 (plot-string-italic (+ 20 width) 15 "Category 2")
 (plot-string-italic (+ 20 width width) 15 "Category 3")
 (dolist (p *learned-categories*)
  (print (list "inputs:" (car p) "category:" (cadr p)))
  (let ((pattern (car p))
     (category-code (cadr p)))
   (plot-input
    (+ 20 (* category-code width)) ;; x coord of plot origin
    (+ 40 (* (nth category-code category-counts) 40)) ;; y coord
    pattern)
   (setf (nth category-code category-counts)
       (1+ (nth category-code category-counts))))))
```

7.5 Running the ART2 Program

The example program defines 15 sample patterns (the global variable *testPatterns* is set equal to a list of the sample patterns in the ART2 program). The sample program listing contains three programs that are executed by typing *(ART2-init 5 3)*, *(ART2)* and then *(ART2-postprocess)* in your Lisp listener window. Figure 7.4 shows the output graphics produced by function *ART2-postprocess*; the automatic categorization (performed by the *ART2* program) of the fifteen sample input patterns into three categories is illustrated by dividing the fifteen input patterns into three categories ordered in Figure 7.4 by column.

The sample program uses two global variables *nInputs* and *nOutputs* to define the number of values in each input pattern and the number of categories to sort the inputs into. Figure 7.4 shows six input patterns being sorted into category 1, five input patterns into category 2, and four input patterns into category 3.

The following text was captured from the Lisp listener window while executing the example program (user input in bold):

```
? (ART2-init 5 3) ;;  5 inputs, 3 output categories, use default training data
nil
? (ART2)  ;; execute the pattern classifier until user resets
("Inputs:" (0.0 0.4 0.9 0.4 0.0))
("Inputs:" (1.0 0.3 0.0 0.3 1.0))
("Inputs:" (0.75 0.5 0.02 0.05 0.07))
("Vigilance reset ="  1.0538614441512  " Learning cycle ="  5)
("Vigilance reset ="  1.73225324598681  " Learning cycle ="  6)
("Inputs:" (0.02 0.41 0.91 0.36 0.08))
<Aborted by control C>
?(ART2-postprocess) ;; execute the postprocessor
("inputs:"  (0.75 0.5 0.02 0.05 0.07)  "category:"  0)
("inputs:"  (1.0 0.3 0.0 0.3 1.0)  "category:"  1)
("inputs:"  (0.0 0.4 0.9 0.4 0.0)  "category:"  2)
nil
?
```

Since the initial state of the ART2 neural network in our example program is random (the connection weights held in the global variables *wUp* and *wDown* are initialized to small random numbers), the example system will create different categories in different execution runs. Furthermore, often one or two input patterns will seem to

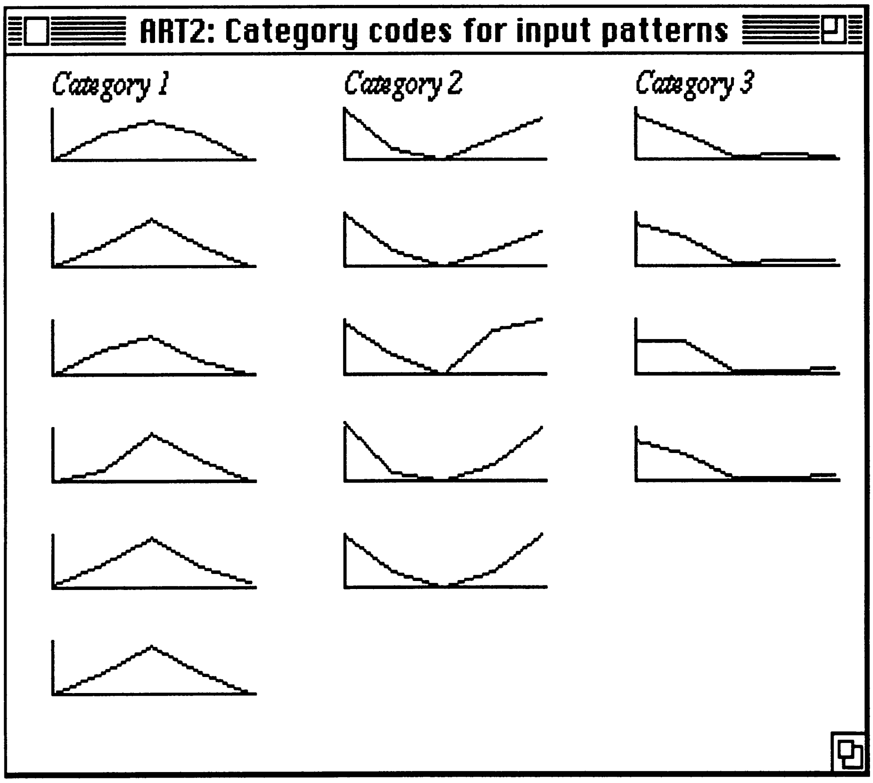


Figure 7.4. Screen dump showing the execution of the ART2 model postprocessor. Automatic categorization of 15 patterns into 3 categories.

be sorted into the "wrong" (different from where most would sort them) category. The ART2 neural network model is very sensitive to the model constants and as stated earlier, changing them is to be avoided. The model constant *vigilance* is the only constant in the sample program which you are likely to change: increasing the value for *vigilance* (maximum value should be 1.0) causes the ART2 network to more exacting in sorting new patterns into existing categories, and decreasing its value (minimum value of 0.0) allows patterns with larger feature differences to be grouped into the same category.

7.6 Project

Choose an application domain that you are familiar with in which input data sets are manually sorted into categories. Without classifying the inputs, use them to train an ART2 neural network. Are the categories created by the ART2 neural network based on the same features you would choose? (For example, if you have a home weather station, try scaling the inputs from your instruments to the dynamic range (0.0, 1.0) and use an ART2 neural network to classify the inputs into a few categories. What features did the neural network use to form the calculated categories?)

8
Representing Natural Language as LISP Data Structures and LISP Code

Natural language processing (NLP) systems allow interaction with a computer using natural language. NLP systems are practical to build when the system vocabulary is small and the domain of discourse limited. In section 8.1, the basic techniques will be developed for parsing Natural Language (NL) text using syntactical analysis by starting with a simple parser which recognizes noun phrases. We will use this first simple noun phrase parser to introduce the notation for augmented transition networks (ATNs).

In section 8.2 we will use both a bubble diagram and LISP notation to write down a parser that can recognize entire sentences. In section 8.3, we will then re-write the parser for *animating* the parsing process by graphically displaying the bubble diagrams while the parser is running. In section 8.4 we will show how to run the animated parser. Section 8.5 presents suggested projects. In chapter 9, we will add semantic processing to syntactic analysis to build a NL interface for a hypothetical library database.

There are two principle approaches to modern NLP systems: syntax directed parsing (Woods, 1968) using ATNs and semantic directed systems such as those using **conceptual dependency** (CD) Theory (Schank and Reisbeck, 1981). CD systems work on language primitives as defined as atomic actions and thoughts (e.g., actions like the transfer of possession and moving an object). CD parsers have a nice feature that they yield the same output for different sentences if the **semantics** (i.e., the meaning) of the sentences is the same. ATN systems are simpler to implement than CD systems, so we will start by using a syntax directed approach for NLP. CD seems to be emerging as a more powerful formalism for systems requiring operation with a less constrained domain of discourse (including un-grammatical input).

8.1 Introduction: Using ATNs to Parse Noun Phrases

ATNs can be expressed using both **bubble diagrams** as in Figure 8.1 and in the next Common LISP program. Bubble diagrams are state transition diagrams in which states are represented as a state name enclosed in an oval and transitions between states are represented as arrows. Transitions usually have an associated condition which must be true for the transition from one state to another to occur. The operation of the noun phrase parser will be demonstrated using lists of words as input to the parser; we will see how the parser removes words from the input list as it recognizes noun phrases. The development of the noun phrase parser is meant to be a tutorial introduction for writing **recursive** parsers (i.e., they can call themselves while parsing).

ATN parsers are pattern matchers for natural language sentences. We will see that this pattern matching is recursive so that the grammar that can be recognized is extensive. We will examine ATNs in two steps: first we will examine a simple ATN to build a conceptual understanding of how they work, then we will develop a more complex ATN which parses complete sentences.

8.1.1 ATN Data Structures

Before presenting ATNs which are capable of recognizing complete sentences, we will examine a small ATN for recognizing noun phrases. Figure 8.1 shows a bubble diagram for an example noun phrase ATN. The ATN diagram in Figure 8.1 has three states: *Start*, *S1*, and *Done*. Transitions can be made between the *Start* state and state *S1* if either of two tests are true: the next word in the input list to the ATN is an adjective, or a "no test" condition; in other words, there can always be a transition between state *Start* and state *S1* in Figure 8.1 because if the first word is not an adjective, the second test is always true (no condition). ATN parsers are recursive; notice in Figure 8.1 that ATN *np?* calls itself (arcs between nodes represent *if* tests which can be calls to ATN networks) via the top arc between state *S1* and *Done*. There can be a transition between state *S1* and state *Done* if either the recursive call to the *np?* network returns a true value, or if the next word in the input list is a noun. If the *np?* network reaches the state *Done*, then the network returns a true value, otherwise it returns a false value. ATNs are **augmented** by arbitrary program code that is executed on successful completion of the *if* tests represented by arcs; these code fragments typically record the current word in a variable.

8.1.2 Implementing ATNs in LISP

ATNs take as input a stream (or list) of words and return a true value if they *recognize* the word stream. The ATN bubble diagram in Figure 8.1 recognizes noun phrases like *'(large dog)*. As an example, if the ATN in Figure 8.1 was called with the input word list *(large dog ran)*, then the *if* test for the top arc between state *Start* and state *S1* would be satisfied since the first input word is an adjective. In

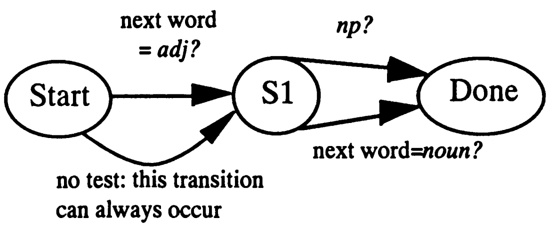


Figure 8.1 ATN Bubble Diagram Demonstrating the Recognition of Noun Phrases

transitioning to state *S1*, the first word would be removed from the input list. Note that if the first input word was not an adjective, the transition between *Start* and *S1* would not "eat" a word since arcs without conditional *if* tests by convention do not remove words from the input list.

ATN bubble diagrams can also be easily represented in LISP programming notation, as seen in the following program listing (again, note the recursion in function *np?*). The function *word?* checks the next word in the input stream (first element of list **sentence**) for being a specified word type. The recursive function *np?* implements the ATN shown in Figure 8.1.

```
;; Example ATN parser: recognize noun phrases

;; Word desciminators: use global noun (*nouns*) and adjective (*adjs*)
;; lists to check if a given word is a noun or adjective:

(defun isNoun? (word)  (member word *nouns*))

(defun isAdj? (word)     (member word *adjs*))

;;;;;; Test for word type:
(defun word? (type)
   (let ( (w (if (null *sentence*) nil (car *sentence*))) ; next word in sentence
            (func  ; the function definition of a word discriminator
             (assoc
              type
              '((noun isNoun?) (adj isAdj?))))) ; set func to isAdj? of isNoun?
      (if func ; see if we found a word type discriminator for this word
         ; apply the word discriminator to the word (eg. (isAdj? w))
        (if (funcall (cadr func) w)
          (let () ; word is of correct type, so remove it from word stream
             (setq *sentence* (cdr *sentence*))
             t) ; .. and return the value of true for the value of this function
          nil) ; return value of false
        nil))) ; return value of false

;;;;;; Parsing network for a noun phrase:
(defun np? ()
 ; save the current word stream in case of failure to match a np?
 (let ((save *sentence*))
   (if
     ; succeed if: 1. adj followed by np, or:  2. current word in word stream is a noun
    (or
     (and ; case: 1. adj followed by np
      (word? 'adj)
      (or (np?) (word? 'noun)))
    (word? 'noun)) ; case: 2. current word in word stream is a noun
   t ;; accept this word stream as a noun phrase
   (let () ; failed test: return word stream to its original state and return nil
     (setq *sentence* save)
     nil))))

;; Define test nouns and adjectives:
(setq *nouns* '(block dog))
(setq *adjs* '(blue green large red the))
```

The following output listing shows this simple ATN being tested by defining the word stream variable **sentence** and executing function *np?*:

```
? (setq *sentence* '(large dog))  ;; initialize the word stream
(large dog)
? (np?)  ;; execute the np? ATN
t
? *sentence*   ;; print out the contents of the word stream
nil
? (setq *sentence* '(large dog ran))  ;; initialize the word stream
(large dog ran)
? (np?)  ;; execute the np? ATN
t
? *sentence*   ;; print out the contents of the word stream
(ran)
? (setq  *sentence* '(I saw the large dog))  ;; initialize the word stream
(i saw the large dog)
? (np?)  ;; execute the np? ATN
t
? *sentence*   ;; print out the contents of the word stream
(saw the large dog)
? (setq *sentence '(saw the large dog))  ;; initialize the word stream
(saw the large dog)
? (np?)  ;; execute the np? ATN (will fail on this example)
nil
?
```

8.2 Parsing English Sentences Using ATNs

To parse complete sentence, we will use the ATNs developed in section 8.1 to parse noun-phrases and add additional networks for parsing prepositional-phrases, verb-phrases, and simple sentences.

ATNs are very easy to implement once the bubble diagrams are designed. Some systems will automatically compile the data from ATN diagrams into LISP or C code. Each of the connected diagrams in Figure 8.2 should be thought of as a separate ATN. There is a strong correspondence between an ATN and a function in a procedural programming language like LISP as seen in the example in Figure 8.1 and the corresponding Lisp code in the last program listing.

The following complete program listing implements ATNs for a simple grammar to recognize complete sentences.

The functions *isNoun?*, *isVerb?*, *isAdj?*, and *isPrep?* check a word for being a noun, verb, adjective, or preposition.

The functions *np?*, *prepp?*, *vp?*, and *sentence?* recognize noun phrases, prepositional phrases, verb phrases and sentences respectively.

The function *word?* is called with a word type and checks the next word in the input

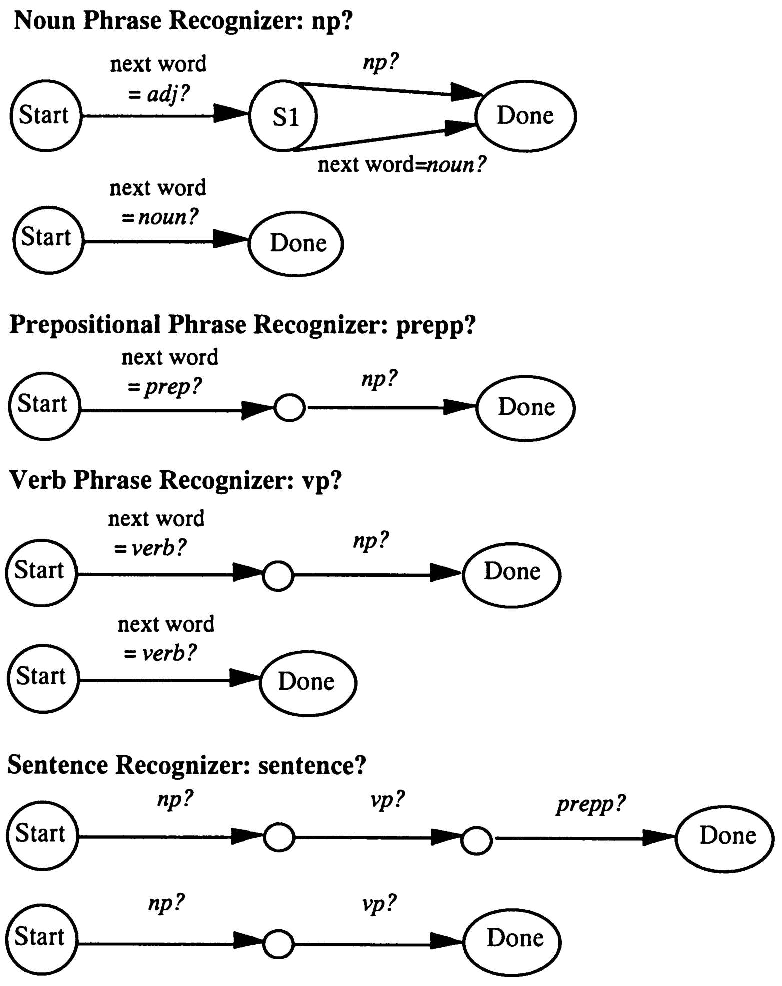


Figure 8.2. "Bubble diagram" for an Augmented Transition Network (ATN) parser

list **sentence** by applying one of the appropriate functions: *isNoun?*, *isVerb?*, *isPrep?*, or *isAdj?*. If *word?* accepts the next word in the variable **sentence** as being of the same type as its argument, then the global variable **sentence** is modified by removing the first element. The value of **sentence** is set to a list of words at the end of the following listing. The Common LISP *trace* function is used to show the call structure of the parsing functions during execution.

```
;;
;File: ATN.LISP
;
; Copyright 1990 by Mark Watson.
;;

;;
; Word descriminators:
;;

(defun isNoun? (word)
 (if (member word *nouns*)
   (setq *last-noun* word)))

(defun isVerb? (word)
 (member word *verbs*))

(defun isAdj? (word)
 (member word *adjs*))

(defun isPrep? (word)
 (member word *preps*))

;;
; Parsing networks:
;;

(defun np? ()
 (let ((save *sentence*))
   (if
     (or
      (and
       (word? 'adj)
       (or (np?) (word? 'noun)))
      (word? 'noun))
     (let ()
       (if (null *subject*) (setq *subject* *last-noun*))
      t)
     (let ()
      (setq *sentence* save)
      nil))))

(defun prepp? ()
  (let ((save *sentence*))
    (if (and
         (word? 'prep)
         (np?))
      t
      (let ()
        (setq *sentence* save)
        nil))))

(defun vp? ()
  (let ((save *sentence*))
    (if (or
         (and (word? 'verb) (np?))
         (let ()
           (setq *sentence* save)
           (word? 'verb)))
      t
      (let ()
        (setq *sentence* save)
        nil))))

(defun sentence? ()
  (let ((save *sentence*))
    (if (or
         (and
          (np?)
          (vp?)
          (prepp?))
         (let ()
           (setq *sentence* save)
           (and (np?)
                (vp?))))
      t
      (let ()
        (setq *sentence* save)
        nil))))
```

```
;;
; Test for word type:
;;

(defun word? (type)
  (let ((save *sentence*)
        (w (if (null *sentence*) nil (car *sentence*)))
        (func
         (assoc
          type
          '((noun isNoun?) (verb isVerb?) (prep isPrep?) (adj isAdj?)))))
    (if func
      (if (funcall (cadr func) w)
        (let ()
          (setq *sentence* (cdr *sentence*))
          t)
        nil)
      nil)))

;;
; Test words:
;;

(setq *nouns* '(block dog I))
(setq *verbs* '(grab see))
(setq *adjs* '(blue green large red the))
(setq *preps* '(behind under))

(setq *subject* nil)

;;
; Test sentence:
;;

(setq *sentence* '(I see the large red block under the large dog))

;;
; Turn on tracing:
;;
(trace isNoun? isVerb? isAdj? isPrep? np? prepp? vp? sentence?)

;;
; Run system:
;;
(sentence?)
```

8.3 Animated ATN Program

The program listed in this section implements an *animatedATNparser* which draws network diagrams and animates them during the parsing process. This animated example ATN shows the recursive nature of parsing networks: they show the "calling" of a given ATN by increasing the line thickness of the bubbles in the animated ATN by one pixel; when we "return from" an ATN, we will reduce the line thickness by one pixel.

The functions *isNoun?*, *isVerb?*, *isAdj?*, *isPrep?*, *np?*, *prepp?*, vp?, and *sentence?* have been modified from the example program in section 8.2 to re-draw the animated bubble diagrams in a graphics window using line thickness in the plot to indicate entry into a parsing function.

The following listing shows the program used to generate the animated ATN diagrams that will be seen in Figures 8.3 and 8.4.

```
;;
; File: ATNanimation.LISP
;
; To execute this sample program, load this file into your
; Lisp workspace after loading the plotting library. Then
; execute the function test. This example program plots sample
; ATN networks graphically and illustraies the execution of the
; networks by hilighting each node in the network when the
; incoming arc (if test) accepts the next word in the sentence.
;
;    Copyright 1989 by Mark Watson
;;


;;
; Word desciminators:
;;
(defun isNoun? (word)
 (if (member word *nouns*)
  (setq *last-noun* word)))

(defun isVerb? (word)
 (member word *verbs*))

(defun isAdj? (word)
 (member word *adjs*))

(defun isPrep? (word)
 (member word *preps*))
```

```
;;
 ; Parsing networks:
 ;;

(setq *s1-np2?-thickness* 1)
(setq *done-np2?-thickness* 1)
(setq *start-np2?-thickness* 1)

(defun np? ()
  (let ((save *sentence*)
        (save-s1 *s1-np2?-thickness*)
        (save-done *done-np2?-thickness*)
        (save-start *start-np2?-thickness*))
   (draw-start-np? 1)
   (if
    (and
     (if (word? 'adj)
       (let ()
        (draw-s1-np? 1)
        (draw-start-np? -1))
       (let ()
        (draw-start-np? -1)
        nil))
     (if (np?)
       (let ()
        (draw-done1-np? 1)
        (draw-s1-np? -1)
        (ATN-delay)
        (draw-done1-np? -1))
       (let ()
        (draw-start-np? -1)
        (draw-s1-np? -1)
        (setq *sentence* save
              *s1-np2?-thickness* save-s1
              *done-np2?-thickness* save-done
              *start-np2?-thickness* save-start)
        nil)))
    T
    (let ()
     (draw-start-np2? 1)
     (if (word? 'noun)
       (let ()
        (draw-done1-np2? 1)
        (draw-start-np2? -1)
        (ATN-delay)
```

```
          (draw-done1-np2? -1))
         (draw-start-np2? -1))))))

(defun prepp? ()
  (let ((save *sentence*))
    (if (and
          (word? 'prep)
          (np?))
      t
      (let ()
        (setq *sentence* save)
        nil))))

(defun vp? ()
  (let ((save *sentence*)
        (save-s1 *s1-vp?-thickness*)
        (save-done *done-vp?-thickness*)
        (save-start *start-vp?-thickness*))
    (draw-start-vp? 1)
    (if (or
          (and
           (if (word? 'verb)
             (let ()
               (draw-s1-vp? 1)
               (draw-start-vp? -1)))
           (if (np?)
             (let ()
                 (draw-done1-vp? 1)
                 (draw-s1-vp? -1)
                 (setq *s1-vp?-thickness* save-s1
                       *done-vp?-thickness* save-done
                       *start-vp?-thickness* save-start)
                 (ATN-delay)
                 (draw-done1-vp? -1)
                 T)))
          (let ()
            (setq *sentence* save
                  *s1-vp?-thickness* save-s1
                  *done-vp?-thickness* save-done
                  *start-vp?-thickness* save-start)
            (if (word? 'verb)
              (let ()
                (draw-done1-vp? 1)
                (setq *s1-vp?-thickness* save-s1
                      *done-vp?-thickness* save-done
                 *start-vp?-thickness* save-start)
            (draw-start-vp? -1)
            (ATN-delay)
            (draw-done1-vp? -1)))))
    t
    (let ()
      (setq *sentence* save
           *s1-vp?-thickness* save-s1
           *done-vp?-thickness* save-done
           *start-vp?-thickness* save-start)
      nil))))

(defun sentence? ()
  (let ((save *sentence*))
    (draw-start 1)
    (if (or
         (and
          (if (np?)
            (let ()
              (draw-s1 1)
              (draw-start -1)))
          (if (vp?)
            (draw-s2 1)
            (let ()
              (draw-s1 -1)
              (draw-s2 1)))
          (if (prepp?)
              (draw-done1 1)
              (draw-s2 -1)))
         (let ()
           (setq *sentence* save)
           (and (np?)
                (vp?))))
      (let ()
        (draw-done1 1)
        (draw-s2 -1)
        (draw-done1 -1)
        (draw-s1 -1)
        T)
      (let ()
        (setq *sentence* save)
        nil))))

;;
; Test for word type:
 ;;
(defun word? (type)
  (let ((save *sentence*)
        (w (if (null *sentence*) nil (car *sentence*)))
        (func
         (assoc
          type
          '((noun isNoun?) (verb isVerb?)
            (prep isPrep?) (adj isAdj?)))))
    (princ "Processing word: ") (princ w)
    (princ "  sentence: ") (princ *sentence*) (terpri)
    (if func
      (if (funcall (cadr func) w)
        (let ()
          (setq *sentence* (cdr *sentence*))
          t)
        nil)
      nil)))

;;
 ; Define test words:
 ;;
(setq *nouns* '(block dog I))

(setq *verbs* '(grab see))

(setq *adjs* '(blue green large red the))

(setq *preps* '(behind under))

(setq *subject* nil)

(setq *last-noun* nil)

;; turn on tracing:

(trace isNoun? isVerb? isAdj? isPrep? np? prepp? vp? sentence?)

;;
 ; Run the system:
 ;;
(setq nodeSizeX 40)
(setq nodeSizeY 20)

(defun plot-node (name x y thickness)
 (ATN-delay)
 ;; erase old node:
 (plot-fill-rect x y nodeSizeX nodeSizeY 0)
 (plot-string (+ x 8) (+ y nodeSizeY -7) name)
 (if (< thickness 1) (setq thickness 1))
 (dotimes (n thickness)
  (let ((xmin (+ x n))
       (ymin (+ y n))
       (xsize (- nodeSizeX (* n 2)))
       (ysize (- nodeSizeY (* n 2))))
    (plot-frame-rect xmin ymin xsize ysize))))

(defun plot-arrow (x1 y1 x2 y2)
 (plot-line x1 y1 x2 y2)
 (plot-line x2 y2 (- x2 4) (- y2 4))
 (plot-line x2 y2 (- x2 4) (+ y2 4)))


;;
 ; Animation code for sentence? network:
 ;;
(setq *s1-thickness* 1)
(setq *s2-thickness* 1)
(setq *done-thickness* 1)
(setq *start-thickness* 1)

(defun draw-start (delta)
 (setq *start-thickness* (+ *start-thickness* delta))
 (plot-node "sent?" 35 20 *start-thickness*)
 T)

(defun draw-s1 (delta)
 (setq *s1-thickness* (+ *s1-thickness* delta))
 (plot-node 's1 85 20 *s1-thickness*)
 T)

(defun draw-s2 (delta)
 (setq *s2-thickness* (+ *s2-thickness* delta))
 (plot-node 's2 135 20 *s2-thickness*)
 T)

(defun draw-done1 (delta)
 (setq *done-thickness* (+ *done-thickness* delta))
 (plot-node 'done 185 20 *done-thickness*)
 T)
```

```
(defun draw-sentence ()
  (setq *s1-thickness* 1)
  (setq *s2-thickness* 1)
  (setq *done-thickness* 1)
  (setq *start-thickness* 1)
  (plot-arrow 46 27 85 27)
  (plot-arrow 100 27 135 27)
  (plot-arrow 150 27 185 27)
  (draw-start 0)
  (draw-s1 0)
  (draw-s2 0)
  (plot-string-italic 70 15 "np?")
  (plot-string-italic 120 15 "vp?")
  (plot-string-italic 165 15 "prepp?")
  (draw-done1 0))

;;
; Animation code for np? network number 1:
;;
(setq *s1-np?-thickness* 1)
(setq *done-np?-thickness* 1)
(setq *start-np?-thickness* 1)


(defun draw-start-np? (delta)
  (setq *start-np?-thickness* (+ *start-np?-thickness* delta))
  (plot-node "np?" 35 60 *start-np?-thickness*)
  T)

(defun draw-s1-np? (delta)
  (setq *s1-np?-thickness* (+ *s1-np?-thickness* delta))
  (plot-node 's1 85 60 *s1-np?-thickness*)
  T)

(defun draw-done1-np? (delta)
  (setq *done-np?-thickness* (+ *done-np?-thickness* delta))
  (plot-node 'done 135 60 *done-np?-thickness*)
  T)

(defun draw-np? ()
  (setq *s1-np?-thickness* 1)
  (setq *s2-np?-thickness* 1)
  (setq *done-np?-thickness* 1)
  (setq *start-np?-thickness* 1)
```

```
 (plot-arrow 46 67 85 67)
 (plot-arrow 100 67 135 67)
 (plot-string-italic 68 55 "adj?")
 (plot-string-italic 120 55 "np?")
 (draw-start-np? 0)
 (draw-s1-np? 0)
 (draw-done1-np? 0))

;;
 ; Animation code for np? network number 2:
 ;;
(setq *s1-np2?-thickness* 1)
(setq *done-np2?-thickness* 1)
(setq *start-np2?-thickness* 1)


(defun draw-start-np2? (delta)
 (setq *start-np2?-thickness* (+ *start-np2?-thickness* delta))
 (plot-node "np?" 35 100 *start-np2?-thickness*)
 T)

(defun draw-s1-np2? (delta)
 (setq *s1-np2?-thickness* (+ *s1-np2?-thickness* delta))
 (plot-node 's1 85 100 *s1-np2?-thickness*)
 T)

(defun draw-done1-np2? (delta)
 (setq *done-thickness* (+ *done-thickness* delta))
 (plot-node 'done 85 100 *done-np2?-thickness*)
 T)

(defun draw-np2? ()
 (setq *s1-np2?-thickness* 1)
 (setq *s2-np2?-thickness* 1)
 (setq *done-np2?-thickness* 1)
 (setq *start-np2?-thickness* 1)
 (plot-arrow 46 107 85 107)
 (plot-string-italic 65 95 "noun?")
 (draw-start-np2? 0)
 (draw-done1-np2? 0))

;;
 ; Animation code for vp? network:
 ;;
(setq *s1-vp?-thickness* 1)
```

```
(setq *done-vp?-thickness* 1)
(setq *start-vp?-thickness* 1)


(defun draw-start-vp? (delta)
  (setq *start-vp?-thickness* (+ *start-vp?-thickness* delta))
  (plot-node "vp?" 35 140 *start-vp?-thickness*)
  T)

(defun draw-s1-vp? (delta)
  (setq *s1-vp?-thickness* (+ *s1-vp?-thickness* delta))
  (plot-node 's1 85 140 *s1-vp?-thickness*)
  T)

(defun draw-done1-vp? (delta)
  (setq *done-vp?-thickness* (+ *done-vp?-thickness* delta))
  (plot-node 'done 135 140 *done-vp?-thickness*)
  T)

(defun draw-vp? ()
  (setq *s1-vp?-thickness* 1)
  (setq *s2-vp?-thickness* 1)
  (setq *done-vp?-thickness* 1)
  (setq *start-vp?-thickness* 1)
  (plot-arrow 46 147 85 147)
  (plot-arrow 100 147 135 147)
  (plot-string-italic 68 135 "verb?")
  (plot-string-italic 120 135 "np?")
  (draw-start-vp? 0)
  (draw-s1-vp? 0)
  (draw-done1-vp? 0))


;;
 ; Delay function to slow down the animation
 ; (ADJUST FOR THE SPEED OF YOUR MACHINE):
 ;;
(setq *ATN-delay* 185000)
(defun ATN-delay ()
 (dotimes (delay *ATN-delay*) nil))

;;
 ; Initialize graphics:
 ;;
(defun init-diagrams ()
  (setq *ATN-delay* 0)
  (init-plot "Animated ATN" 290 200)
  (draw-sentence)
  (draw-np?)
  (draw-np2?)
  (draw-vp?)
  (setq *ATN-delay* 80000)) ; set delay for your machine !!

;;
; MAIN test function:
;;
(defun test ()
  ;; Reset the test sentence:
  (setq *sentence*
       '(I see the large red block under the large dog))
  (init-diagrams)
  (sentence?))
```

8.4 Running the Animated ATN Parser

By using the LISP *trace* function, the recursive nature of the ATN parsing process becomes apparent. Notice that the LISP functions which simulate arcs in our ATN diagrams (e.g., *isVerb?*) consume words in the sentence being parsed if their test conditions are met and otherwise fail (return the LISP value *nil*), returning the sentence to the same state that it was in when the function was first entered.

We now will run the modified version of the above ATN parser to develop a better understanding of the recursive parsing process. When we execute the animated ATN parser, it is necessary to set the delay in function *delay* to a sufficiently large value so that the animation executes slowly so the animation is visible.

Figures 8.3 and 8.4 show the execution of the animated ATN example program. This program plots four ATNs: *sent?*, *np?*, *np?* (noun only), and *vp?*. When an ATN network is entered (i.e., the LISP function which implements the network is called), the first node in the network is highlighted. The arcs between nodes are implemented as *if* tests which check the next word or group of words in the sentence being parsed; this check usually involves calling another ATN. In the screen image in Figure 8.3, the ATN *sent?* has been called, which in turn has called the ATN *np?*. The ATN *np?* has been successfully been traversed because the second node S1 in ATN *sent?* is highlighted. The screen image was captured while ATN *vp?* is executing (being called from ATN *sent?*).

The screen image in Figure 8.4 was captured after the ATN *sentence?* successfully parsed the example sentence.

This ATN parser stops when it finds one path though its subnetworks which legally parses a sentence. Production ATN parsers often produce dozens of legal parses

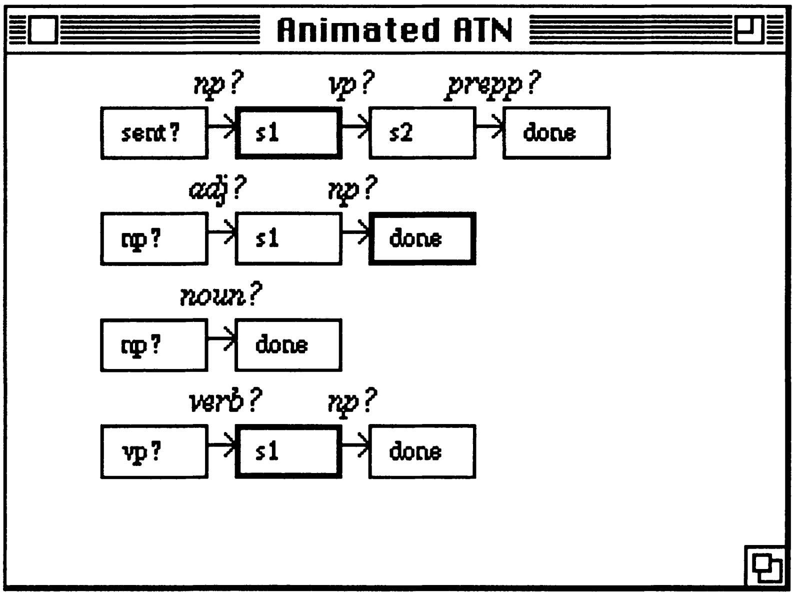


Figure 8.3. Animated ATN Diagram. Shows the sentence "I see a large red block under the large dog." being parsed. The words "I see a large red" have already been processed.

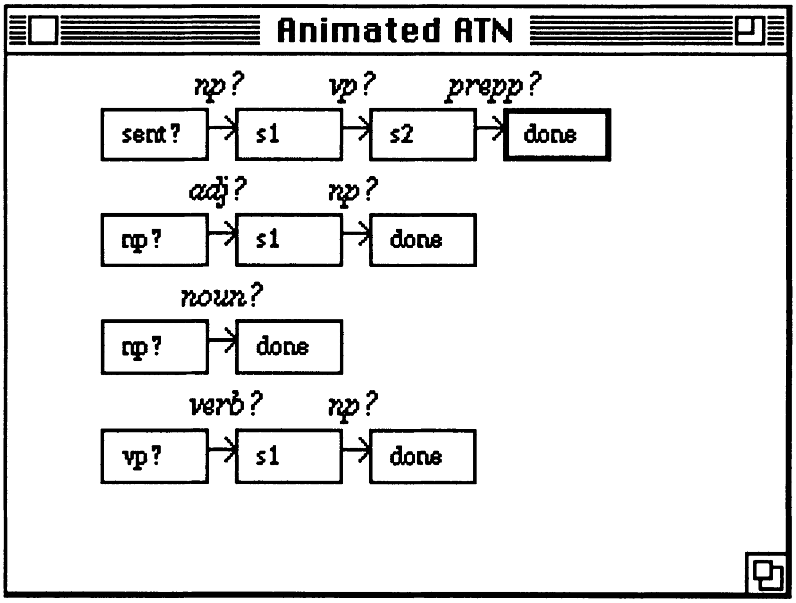


Figure 8.4. Animated ATN after Successful Parse of the Test Sentence.

from a single sentence; the ensuing processing which eliminates all but one parse tree is difficult and involves the use of semantic processing.

Handling ambiguity in NLP is one of the central problems in computational linguistics. The ATN parser that is developed in chapter 9 is similar to conceptual dependency parsers in that it uses semantic processing during the parsing process to both collect the semantics of the processed sentence and to avoid ambiguity by reducing the number of legal parses output by the parser.

8.5 Projects

1. Using the sentence parser from section 8.2, experiment with increasing the vocabulary by adding more nouns, verbs and adjectives.This is a simple exercise until you start to add words which can be different parts of speech depending on context (for example, consider the word "vote").

2. In experimenting with the sentence parser, note sentences that are not recognized. Modify the LISP ATN code to accept a new type of sentence structure. Make sure that the new parser also recognizes all previous test cases that the previous version of the parser recognized. In developing NLP parsers, it is useful to have a large library of test sentences and set up the parser to process every sentence in the library; this development aid will save much typing!

9
Natural Language Interface to a Library Database

For many software systems (e.g., computer aided design and computer aided software engineering tools), high-resolution graphical windowing interfaces are the appropriate user interface technology. However, for systems like a library database, inexpensive "dumb" terminals that accept natural language (NL) commands and that respond in natural language provide a much more usable interaction for those who care most about getting their data management tasks done quickly. Clearly, natural language processing systems augmented by reliable speech recognition systems will, in the near future, provide the most natural possible man-machine interface for low-bandwidth interactions.

In this chapter we will design and build a database management tool with a natural language interface tailored specifically to the application domain of keeping track of books in a library. We will discuss design, data integrity requirements, and the grammar necessary for understanding natural language commands to update and query information in the database. Our primary design goal is to build a database system controlled exclusively by simple natural language commands.

The NLP program in this chapter deals with **semantics** (or meaning) of processed text. In chapter 8, the NLP program performed syntactical processing of text. We will use the sentence parser developed in chapter 8 and add a layer of software for extracting semantic information.

9.1 Design for Library Database Natural Language Interface

We want to design an interface to a database system with the following features:

- Support for natural language queries
- Frame data representation
- Modular system design
- Efficient operation in a LISP environment
- No requirement for an external database system

9.1.1 Natural Language Queries

In designing a grammar for parsing simple domain-specific natural language commands and queries, it often helps to write down some examples of the type of text we would like our system to be able to handle. The following are examples of the range of natural language interaction our system will support:

> List all books by Yukio Mishima.
> Show books by Mishima.
> Mark Watson checked out "Colossus and the Crab".
> Julie Kimmel got "The Sound and the Fury".
> "The Sound and the Fury" was returned by Mark Watson.

The parser will need help in determining what parts of the queries are book titles. For this reason, interactions with the NL interface will require that book titles be typed in as strings (that is, embedded in quotation marks). Note that this grammar is extended from the grammar used in chapter 8 (shown in Figure 8.2) in that a sentence can also be a verb phrase followed by a noun phrase. This extended grammar allows the system to parse commands beginning with verbs. All of the ATN parsing functions are similar to those developed in the last chapter, with the addition of code to extract the semantic information.

You can test a parsing system by having independent testers try different sentences. Record sentences which are not correctly parsed (or not parsed at all) and iteratively augment the ATN parser until it can parse a sufficiently large percentage of test sentences in the target domain.

Also bear in mind that not all reasonable sentences must be processed by a NL system. Since a NL system usually paraphrases the perceived semantic content of commands back to users, failure of the system is instantly noted by users and they can rephrase their request. The example program in Section 9.2 replies to user requests for information by paraphrasing the query and then listing the requested information. In practice, users quickly learn what is an acceptable grammar for interacting with a given NL system to the extent that we can reasonably expect user adaptation to a restricted grammar and domain of discourse to occur on almost a subliminal level.

9.1.2 Frame Data Representation

Semantic information in this system will be kept in frame data structures. A **frame** is a single data element with an internal structure consisting of one or more attribute/value pairs. Consider the following data definition:

```
(setq test-frame
    '((type 'command)
      (action 'list-info)
      (object 'book)
      (author 'faulkner)))
```

The definition of the variable *test-frame* defines a list in which each sublist is an attribute/value pair. To retrieve the value of the attribute *auhor*, the system searches each sublist until it finds a list in which the first element is equal to *author*. The second element in the sublist is the desired value for *author*.

9.1.3 Modular System Design

The bubble diagram in Figure 9.1 shows both the design for information and control flow in the natural language interface developed in this chapter. It should be noted that this high-level design does not depend specifically on our library data base application.

9.1.4 Efficient Operation

So that our system can efficiently handle very large databases quickly accessing both books stored in the library and authors, we will use two **hash tables** to reference the two semantic structure formats used in the system. Hash tables are a mechanism for quickly locating data items given a key or index value. Most Common LISP

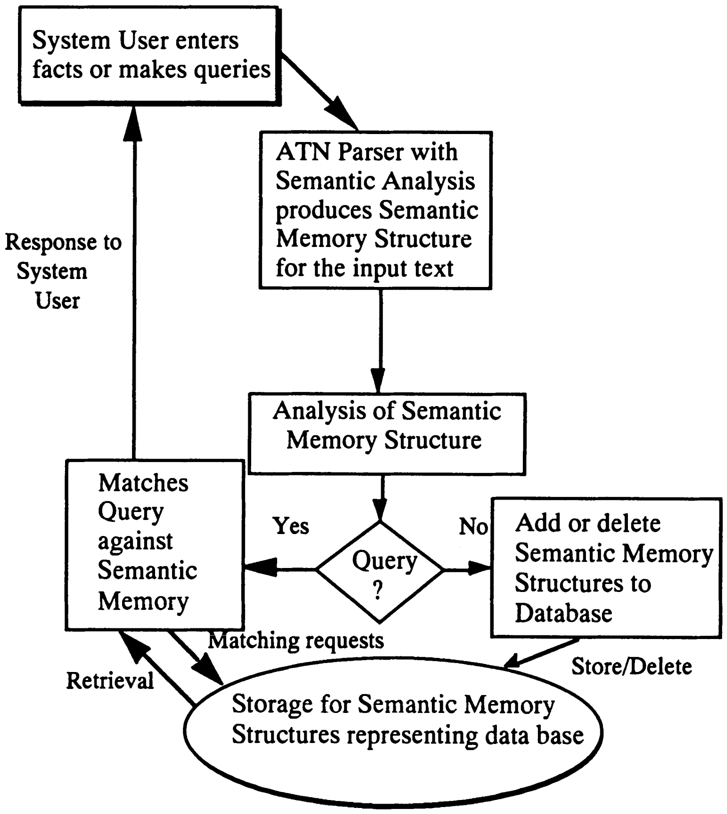

System User enters facts or makes queries
ATN Parser with Semantic Analysis produces Semantic Memory Structure for the input text
Response to System User
Analysis of Semantic Memory Structure
Matches Query against Semantic Memory
Yes
Query ?
No
Add or delete Semantic Memory Structures to Database
Matching requests
Retrieval
Store/Delete
Storage for Semantic Memory Structures representing data base

Figure 9.1. Design for a Natural Language Interface to a Library data base system

implementations have very efficient hashing mechanisms. For readers using older dialects of LISP which do not support hash tables directly in the language, it is left as an exercise to implement the functions *make-hash-table* and *gethash* (note that *gethash* must work as an argument to *setf*).

The organization for authors is obvious: we will use the author's last name to hash into a list containing semantic memory structures for everyone with this same last name. To calculate the hashing index for books, we will use the first word of the book title which is neither an article or a preposition and all characters in the title will be mapped to upper case. For example, the book title The Sound and the Fury would be stored using a hash index of *"SOUND AND FURY"*.

9.1.5 Use of LISP Data Dtructures Instead of an External Database

In our library database application, the information for books and authors is stored in the same format as the semantic memory structures that result from parsing NL input but it is represented as frames. Data modification natural language commands (for example, *The book "The Sound and the Fury" was check out by Betsy Forest*) modify the semantic memory structures which make up our database. There are just two frame memory structures used in this system: one to represent books and the other to represent people with library cards.

To make our demonstration system usable in a production mode, we need the capability to save and restore semantic memory to and from disk. This symbolic storage of the system data is done in case the system data (LISP memory structures) become corrupted. For Common LISP environments with the capability of dumping the entire LISP image to disk, a production system could work by keeping three dump image files on disk: these dump files will be called the grandfather, father, and son dump files. In a production environment, these files could be rotated to provide a secure backup system. A further requirement for data integrity which must be reflected in our design is an incremental logging of changes to semantic memory between system dumps to protect the system from machine crashes. An excellent fail-safe logging mechanism could also be a low-cost printer which logs all data changes in a terse format; in the eventuality of a system crash (for example, due to a power failure), the database could be manually updated back to the point in time when the last system save was performed.

It is important to design in data integrity and backup procedures into a database system. Fortunately, it is very easy to handle file input/output in Common LISP, so that adding symbolic file save, data logging, and systematic backup functionality into our system would require very few lines of code. Symbolic file save code is not in the example program in section 9.2. Different Common LISP environments have built in capabilities for saving both individual data objects and the entire LISP environment. Although we usually try to write totally portable Common LISP code, implementation specific data storage facilities should probably be used for efficiency in this case. The following output from a LISP listener window does show a portable, if not efficient, way to save LISP data objects to external data files:

```
? (setq x1 '(1 (2 (3 4 5 6))))    ;; example list
(1 (2 (3 4 5 6)))
? (setq x2 '(the cat (ate the mouse (quantity 0))))    ;; example list
(the cat (ate the mouse (quantity 0)))
? (setq x-lists (list x1 x2))   ;; example list to save to disk
((1 (2 (3 4 5 6)))
 (the  cat  (ate the mouse (quantity 0))))
? (with-open-file (out "test.dat" :direction :output)  ;; write to file  "test.dat"
     (print x-lists out))
((1 (2 (3 4 5 6)))
 (the  cat  (ate the mouse (quantity 0))))
? (setq
   new-list
   (with-open-file (in "test.dat" :direction :input)  ;; read from file "test.dat"
      (read in)))
((1 (2 (3 4 5 6)))
 (the  cat  (ate the mouse (quantity 0))))
? new-list
((1 (2 (3 4 5 6)))
 (the  cat  (ate the mouse (quantity 0))))
? (car new-list)
(1 (2 (3 4 5 6)))
? (cadr new-list)
(the cat (ate the mouse (quantity 0)))
?
```

9.2 Example Natural Language Interface Program

In the example program in this section, function *parse* processes the list of words passed to it as a function argument. Function *parse* first initializes global variables used as semantic processing registers; for example the global variable **book-list** holds the name of any book title encountered in the syntactic parsing process. The variable **action** holds the semantic query type extracted from the input sentence; the possible values are *list-info*, *return-book*, and *check-out-book*. Function *parse* calls the ATN parsing function *sentence?* (similar to the function *sentence?* defined in the last chapter with additional code to extract semantic information). After parsing the sentence and extracting semantic information, function *parse* uses the value of the variable **action** to respond to the user's query or statement. Our system performs some "preprocessing" on input text (for example, it converts names to a canonical list form: (last first middle)).

The functions *pp-author* and *pp-book* are utility functions to print out English responses for queries on author and book information. The functions *isNoun?*, *isVerb?*, *isAdj?*, *isPrep?*, and *isBook?* are utility functions to determine word types (a book title is treated as a single token by the parser).

The ATN parsing functions are similar to those defined in the last chapter with the addition of code to extract semantic information. The ATN parsing functions are *np?*, *prepp?*, *vp?*, *sentence?*, and *card-holder-name?*. As these ATN functions recognize word streams, they set the semantic parsing registers. The ATN function *card-holder-name?* accepts a word stream as being a card holder name if the word stream starts with a legal name in the global list **library-card-holders**. Function *word?* is passed a word type (values of: noun, verb, prep, or adj) and calls the word type discriminator function for the requested word type.

The sample database is defined initially in the global lists **book-list** and **library-card-holders**. The function *init-database* creates two hash tables for books and authors and initializes them by inserting author and book names.

The following listing shows the library database system:

```
;;
; File: LibraryATN.Lisp
;
; This is a natural language interface for a library database system
; Copyright 1990 by Mark Watson
;;

(proclaim '(special *library-card-holders* *sentence* *author-hash-table*))

;;
; Main interface function:
;  Parse(sentence) — sentence is the sentence to process
;                  — returned value is a semantic memory structure
;;

(defun parse (sentence &aux original-sentence)
  (setq *subject* nil
        *verb* nil
        *book-title* nil
        *object-1* nil
        *modifier-1* nil
        *number-1* nil
        *modifier-type-1* nil
        *card-holder* nil
        *action* nil
        *last-noun* nil
        *last-verb* nil
        *last-prep* nil
        *last-word* nil
        *response* nil
        *response-type* nil)
  (setq *sentence* sentence original-sentence (copy-list sentence))
  (if (not (sentence?)) ;; cheap trick: if you fail to parse,
    (let ()             ;; discard the first word and try again
      (setq *last-word* (car *sentence*))
    (setq *sentence* (cdr *sentence*))
    (sentence?)))
;; if *object-1* has not been set, look for a book title:
(if (null *object-1*)
  (dolist (w original-sentence)
    (if (stringp w)
      (setq *object-1* w))))
(princ "Semantic memory structure:") (terpri)
(princ "  Subject             : ") (princ *subject*) (terpri)
(princ "  Action              : ") (princ *action*) (terpri)
(princ "  Library card holder : ") (princ *card-holder*) (terpri)
(princ "    Action object       : ") (princ *object-1*) (terpri)
(princ "    Object modifier type: ") (princ *modifier-type-1*) (terpri)
(princ "    Object modifier     : ") (princ *modifier-1*) (terpri)
(princ "    Number modifier     : ") (princ *number-1*) (terpri)
(terpri)
;; Use semantics of user prompt to perform data base query
;; or update operation:
(case *action*
  (nil (terpri))
  (list-info
   (if *response*
     (case *response-type*
       (nil (princ "What?") (terpri))
       (author (pp-author *response*))
       (book (pp-book *response*)))))
  (return-book
   (princ "Returning the book: ")
   (princ *object-1*)
   (princ ".") (terpri)
   (princ "Returned by ")
   (dolist (x *card-holder*)
     (princ x)
     (princ " "))
   (princ "and ") (princ *object-1*)
   (princ " is returned to circulation.")
   (terpri))
  (check-out-book
   (princ "Checking out the book: ")
   (princ *object-1*) (princ ".") (terpri)
   (princ "Checked out by confirmed library card holder")
   (terpri)
   (dolist (x *card-holder*)
     (princ x)
     (princ " "))
   (princ "and data base is modified to show") (terpri)
   (princ "that this volume is checked out.")
   (terpri))))
```

```
;;
; Print out information about author:
;;

(defun pp-author (data)
  (princ "The following book(s) are by ")
  (dolist (x  (cadaar data))
    (princ x)
    (if (equal x (car (cadaar data))) (princ ","))
    (princ " "))
  (princ ":") (terpri)
  (dolist (book (car data))
    (princ (car book))
    (terpri)))

;;
; Print out information about a book:
;;

(defun pp-book (data)
  (princ (cadaaar data))
  (princ "is the author.")
  (terpri))

;;
; Word disciminators:
;;

(defun isNoun? (word)
  (if (member word *nouns*)
    (let ()
      (setq *last-noun* word)
      (if (or
           *subject*  ;; already defined
           *last-verb*) ;; must be an imperative
        (setq *object-1* word))
      t)))
(defun isVerb? (word)
  (if (member word *verbs* :test #'equal)
    (let ()
      (if (null *subject*)  ;; assume that sentence is an imperative
        (setq *subject* 'computer))
      ;; semantic hooks to set *action* == 'list-info:
      (if (or
           (member word '(list show))
           (and
            (equal word 'are)
            (equal *last-word* 'what)))
        (setq *action* 'list-info))
      ;; semantic hooks to set *action* == 'return-book
    (if (or
        (member word '(return returned) :test #'equal)
        (and
         (member word '(check checked) :test #'equal)
         (if (> (length *sentence*) 1)
           (equal (cadr *sentence*) '(in back)))))
      (setq *action* 'return-book
           *sentence* (cdr *sentence*)))
    ;; semantic hooks to set *action* == 'check-out-book
    (if (or
        (member word '(got receive received withdrew) :test #'equal)
        (and
         (member word '(check checked) :test #'equal)
         (if (> (length *sentence*) 1)
           (member (cadr *sentence*) '(out) :test #'equal))))
      (setq *action* 'check-out-book
           *sentence* (cdr *sentence*)))
    (setq *last-verb* word))))

(defun isAdj? (word)
  (if (member word *adjs*)
    (if (null *modifier-1*)
      (setq *modifier-1* word)
      t)))

(defun isPrep? (word)
  (if (member word *preps*)
    (let ()
      (setq *last-prep* word)
      (if (member word '(a an one this that))
        (setq *number-1* 1)
        (if (member word '(all every))
          (setq *number-1* 999999)
          t)))))

(defun isBookTitle? (sentenceElement)
  (stringp sentenceElement))

;;
; Parsing networks:
;;

(defun np? ()
  (let ((save *sentence*) temp)
    (if
     (or
      ;; name of member of the library?:
      (and
       ;; imbedded semantics: do not allow parse to generate
       ;; a form like: 'by <author name>' if known action in
      ;; the sentence is 'list-info':
       (not (equal *action* 'list-info))
       (setq temp (card-holder-name?))
       (if (and
            temp
            (null *card-holder*))
         (setq *card-holder* temp)))
     (and
      (word? 'adj)
      (or (np?) (word? 'noun)))
     (word? 'noun))
    (let ()
      (if (null *subject*) (setq *subject* *last-noun*))
      t)
    (let ()
      (setq *sentence* save)
      nil))))

(defun prepp? ()
  (let ((save *sentence*)
        (last-author-name nil)
        (last-card-holder-name nil)
        (temp nil))

    ;;
    ; Define an ATN to detect author names.  Since this ATN is
    ; is only called from inside the 'prepp?' ATN, we will define
    ; its LISP function lexically inside 'prepp?'.
    ;;
    (defun author-name? ()
      (if (gethash (car *sentence*) *author-hash-table*)
        (let ()
          (setq last-author-name (car *sentence*))
          (setq *sentence* (cdr *sentence*)) ;; throw away processed word
          t)  ;; return True: ATN successfully traversed
        ;; check to see if the second word left in the sentence
        ;; is a known author's last name:
        (if (> (length *sentence*) 1) ;; more than 1 word left ?
          (if (gethash (cadr *sentence*) *author-hash-table*)
            (let ()
              (setq last-author-name (cadr *sentence*))
              (setq *sentence* (cddr *sentence*)) ;; throw away processed words
              t)))))  ;; return True: ATN successfully traversed

    (if (or
         (and
          (word? 'prep)
          (np?))
         (and  ;; special check for 'by <author name>'
          (setq *sentence* save)
         (word? 'prep)
         (equal *last-prep* 'by)
         (author-name?)
         (let () ;; we found a form like: 'by <author name>'
           (setq *modifier-type-1* 'by)
           (setq *modifier-1* last-author-name)
           t))
        (and  ;; special check for 'by <card holder name>'
         (setq *sentence* save)
         (word? 'prep)
         (equal *last-prep* 'by)
         ;; imbedded semantics: do not allow parse to generate
         ;; a form like: 'by <author name>' if known action in
         ;; the sentence is 'list-info':
         (not (equal *action* 'list-info))
         (setq temp (card-holder-name?))
         (let () ;; we found a form like: 'by <card holder name>'
           (setq *modifier-type-1* 'by)
           (setq *modifier-1* temp)
           t)))
      t
      (let ()
        (setq *sentence* save)
        nil))))

(defun vp? ()
  (let ((save *sentence*))
    (if (or
         (and (word? 'verb) (np?))
         (let ()
           (setq *sentence* save)
           (word? 'verb)))
      (let ()
        (if (null *action*) (setq *action* *last-verb*))
        t)
      (let ()
        (setq *sentence* save)
        nil))))

(defun sentence? ()
  (let ((save *sentence*))
    (if (or
         (and
          (vp?) ;; imperative
          (prepp?)
          (prepp?))
         (and
          (vp?) ;; imperative
          (prepp?))
         (let ()
          (setq *sentence* save)
          (and
           (np?)
           (vp?)
           (prepp?)))
        (let ()
          (setq *sentence* save)
          (and (np?)
               (vp?)))
        (let ()
          (setq *sentence* save)
          (if (vp?)
            (setq *subject* 'computer) ;; imperative (command)
            nil)))
     t
     (let ()
       (setq *sentence* save)
       nil))))

;;
; Define an ATN to detect card holder names:
;;

(defun card-holder-name? ()
  ;; sentence must be long enough for a first and last name:
  (if (> (length *sentence*) 1)
    (let ((name-list (list (car *sentence*) (cadr *sentence*))))
      (if (member name-list *library-card-holders* :test #'equal)
        (let ()
          (setq last-card-holder-name name-list)
          (setq *sentence* (cddr *sentence*)) ;; throw away processed word
          name-list)))))  ;; return True: ATN successfully traversed

;;
; Test for word type:
;;

(defun word? (type)
  ;; grab semantic information on book/author if appropriate:
  (if (not *response*)
    (if (member *last-prep* '(by))
      (let ((author-data1
              (gethash (car *sentence*) *author-hash-table*))
             (author-data2
              (if (> (length *sentence*) 1)
                (gethash (cadr *sentence*) *author-hash-table*))))
        (if author-data1
          (setq *response-type* 'author
                *modifier-1* (car *sentence*)
                *response*
              (cons author-data1
                   *response*))
        (if author-data2
          (setq *response-type* 'author
               *modifier-1* (cadr *sentence*)
               *response*
               (cons author-data2
                   *response*)))))))
  (let ((save *sentence*)
       (w (if (null *sentence*) nil (car *sentence*)))
       (func
        (assoc
         type
         '((noun isNoun?) (verb isVerb?) (prep isPrep?) (adj isAdj?)))))
    (if func
      (if (funcall (cadr func) w)
        (let ()
          (remove-word)
          t)
        nil)
      nil)))

;;
; Utility to remove first word remaining in sentence AND to
; save the discarded word in *last-word*:
;;

(defun remove-word ()
  (setq *last-word* (car *sentence*))
  (setq *sentence* (cdr *sentence*)))

;;
; Define words in the system:
;;

(setq *nouns* '(I book books library))

(setq *verbs* '(list show are is add was see check checked
              got returned))

(setq *adjs* '(what the a an one this that all every))

(setq *preps* '(in behind under by))

;;
; Turn on tracing for debug output:
;;

(trace isNoun? isVerb? isAdj? isPrep? np? prepp? vp? sentence?
      card-holder-name?)
```

```
;;
; Sample Data Base:
;;

(setq *book-list*
    '(
      ("The Sound And The Fury" (Faulkner William))
      ("The Sound of Rain" (Peebody Alfred))
      ("Broca's Brain" (Sagan Carl))
      ("Dragons of Eden" (Sagan Carl))))

(setq *library-card-holders*
    '(
      (Mark Watson)
      (Carol Watson)
      (Julie Kimmel)
      (David Kimmel)))

;;
; Utilities to pre-process book list to remove prepositions
; Book titles for more robust pattern matching.  Also convert
; from a string to a list of words:
;;

(defun shrink-title (title-string)
  (let (retList)
    (with-input-from-string (aStream (string-upcase title-string))
      (dotimes (i 20)
        (let ((token (read aStream nil nil nil)))
          (if token
            (if (not
                  (member token '(a an the that this)))
              (setq retList (cons token retList)))))))
    (cons 'book\: (reverse retList))))

;;
; Data management functions: Initialize the database by
; shrinking down book titles and building two hash tables:
;
;   1. *book-hash-table*     - key is the first word of the
;                              shrunken book title.
;   2. *author-hash-table*   - key is the author's last name.
;
;   *library-card-holders* =  list of members of the library
;;

(defun init-database ()
  (setq *book-hash-table* (make-hash-table))
  (setq *author-hash-table* (make-hash-table))
```

```
 (dolist (aBook *book-list*)
   (let ((title-hash-key (cadr (shrink-title (car aBook))))
         (author-hash-key (caadr aBook)))
     (setf
      (gethash title-hash-key *book-hash-table*)
      (cons aBook (gethash title-hash-key *book-hash-table*)))
     (setf
      (gethash author-hash-key *author-hash-table*)
      (cons aBook (gethash author-hash-key *author-hash-table*))))))

(init-database) ;; initialize the system when this file is loaded.
```

9.3 Example Execution Runs Of NLP Program

This program uses a syntax directed ATN parser with embedded semantic hooks to capture the user's request. The following listing shows a sample system query:

```
? (parse '(list all books by Sagan))
 Calling (VP?)
  Calling (NP?)
   Calling (ISADJ? ALL)
   ISADJ? returned NIL
  NP? returned NIL
 VP? returned T
 Calling (PREPP?)
  Calling (NP?)
   Calling (ISADJ? BOOKS)
   ISADJ? returned NIL
  NP? returned T
 PREPP? returned T
 Calling (PREPP?)
  Calling (NP?)
   Calling (ISADJ? SAGAN)
   ISADJ? returned NIL
  NP? returned NIL
 PREPP? returned NIL
 Calling (VP?)
 VP? returned NIL
 Calling (NP?)
  Calling (ISADJ? LIST)
  ISADJ? returned NIL
 NP? returned NIL
 Calling (NP?)
```

```
  Calling (ISADJ? LIST)
  ISADJ? returned NIL
 NP? returned NIL
 Calling (VP?)
  Calling (NP?)
   Calling (ISADJ? ALL)
   ISADJ? returned NIL
  NP? returned NIL
 VP? returned T
Semantic memory structure:
  Subject          : COMPUTER
  Action           : LIST
  Library card holder   : NIL
     Action object   : BOOKS
      Object modifier type: BY
     Object modifier : SAGAN
     Number modifier : 999999

The following books are by SAGAN, CARL :
Dragons of Eden
Broca's Brain
NIL
?
```

The sample program file contains a Common LISP *trace* call to turn on the tracing mechanism so that the execution through the individual LISP functions (which are the ATN sub-networks) can be seen. The following example program output shows a brief interaction (with user input typed in **bold characters**) with the NLP system with the LISP function call tracing turned off:

```
? (untrace)
nil
? (parse '(list all books by William Faulkner))
Semantic memory structure:
  Subject             : COMPUTER
  Action              : LIST-INFO
  Library card holder   : NIL
    Action object       : BOOKS
    Object modifier type: BY
    Object modifier     : FAULKNER
    Number modifier     : 999999

The following books are by FAULKNER, WILLIAM :
The Sound And The Fury
NIL
```

```
? (parse '(Show books by Faulkner))
Semantic memory structure:
 Subject           : COMPUTER
 Action            : LIST-INFO
 Library card holder  : NIL
  Action object      : BOOKS
  Object modifier type: BY
  Object modifier    : FAULKNER
  Number modifier    : NIL

The following books are by FAULKNER, WILLIAM :
The Sound And The Fury
NIL
? (parse '(Mark Watson checked out "The Sound and the Fury"))
Semantic memory structure:
 Subject           : COMPUTER
 Action            : CHECK-OUT-BOOK
 Library card holder  : (MARK WATSON)
  Action object      : The Sound and the Fury
  Object modifier type: NIL
  Object modifier    : NIL
  Number modifier    : NIL

Checking out the book: The Sound and the Fury.
Checked out by confirmed library card holder
MARK WATSON and data base is modified to show
that this volume is checked out.
NIL
? (parse '(Mark Watson returned "The Sound and the Fury"))
Semantic memory structure:
 Subject           : COMPUTER
 Action            : RETURN-BOOK
 Library card holder  : (MARK WATSON)
  Action object      : The Sound and the Fury
  Object modifier type: NIL
  Object modifier    : NIL
  Number modifier    : NIL

Returning the book: The Sound and the Fury.
Returned by MARK WATSON and The Sound and the Fury is returned to
circulation.
NIL
? (parse '(List the books by Carl Sagan))
Semantic memory structure:
 Subject           : COMPUTER
 Action              : LIST-INFO
 Library card holder  : NIL
   Action object       : BOOKS
   Object modifier type: BY
   Object modifier     : SAGAN
   Number modifier     : NIL

The following books are by SAGAN, CARL :
Dragons of Eden
Broca's Brain
NIL
? (parse '(David Kimmel got "The Dragons of Eden"))
Semantic memory structure:
 Subject             : COMPUTER
 Action              : CHECK-OUT-BOOK
 Library card holder  : (DAVID KIMMEL)
   Action object       : The Dragons of Eden
   Object modifier type: NIL
   Object modifier     : NIL
   Number modifier     : NIL

Checking out the book: The Dragons of Eden.
Checked out by confirmed library card holder
DAVID KIMMEL and data base is modified to show
that this volume is checked out.
NIL
?
```

Note that the parser knows the names of people holding valid library cards. As seen in the ATN parsing program listing, the parser is augmented with program code to extract and record semantic features. The parsing networks use information in the book, author, and library card holder databases to make decisions on accepting a legal syntactic form (e.g., prepositional phrase). The use of semantic processing in the ATN parser allows us to write a useful system without backtracking or generating multiple hypothesized sentence structures. Probably the largest problem in Computational Linguistics today is the proper handling of the ambiguity which results in the generation of multiple legal syntactical parses from a single sentence. In this example, we front load the parsing process by augmenting it with all of the domain knowledge that we have: our ATN parsing networks only accept sentences that both fit syntactical requirements of the grammar that we implemented and that contain word combinations that contain extractable semantic content. The flavor of the parser is similar to Conceptual Dependency parsers (Schank and Riesbeck, 1981).

9.4 Project

Extend the grammar for this system by adding a new type of query. Start by writing down several example sentences with a common syntactic sentence structure. Assuming that the parser will not accept the grammar of your new query, extend the parser to syntactically accept sentences using your new expanded grammar. Make sure that the parser can still process sentences of the old grammar! Finally, add semantic hooks to your new parser to extract the necessary information from user queries. Modify the output code at the bottom of function *parse* so that users of your new system can see that the program understands their queries.

10
Expert-Systems

Expert-system technology, now widely used as a component of larger software systems, is characterized by the separation of **heuristic knowledge** (i.e., a set of if-then rules) in a program from the rest of the system. An expert-system shell provides a higher level language for expressing problem-solving knowledge in the form of if-then rules. Expert-system construction is facilitated by the use of an existing shell tool. Both public domain and commercial expert-system building shells are available in many programming languages (e.g., LISP, C, Fortran, and Ada). This chapter begins with a definition of terms used to discuss expert-systems, then presents example shells suitable for educational use; they can be extended in your application programs to build a customized high-level programming language environment for adding modular knowledge bases to the systems that you build. Then a backward-chaining expert-system shell is developed which starts with a goal and searches backwards until an initial state is found. This chapter also discusses features to look for in commercial expert-system shells, and recommends two public domain tools now widely used for building expert-systems, then ends with suggested expert-system projects.

The use of expert-system technology is now common practice. Expert-systems are used in such diverse application areas as: oil exploration, business planning, and factory automation. I have participated in the use of embedded expert-system technology for the automatic control of remote computer networks and for tracking targets from incomplete and possibly contradictory sensor data.

Expert-systems are most useful for solving problems for which there is no known algorithmic solution, or if a known algorithm solution is too time consuming. Expert-systems are characterized as providing good solutions to problems but not optimal solutions.

Since the knowledge in expert-systems is kept in separate modules of heuristic rules, it is often useful to use expert-systems in control applications in which the heuristic rules can be quickly modified to tune system performance.

Heuristic rules are statements that we accept as true as the result of experience and that can be generally applied to solve problems. For example, when a car does not start, we apply rules learned through experience for figuring out why the car will not start: "if the starter does not run when we turn the key, then the battery might be dead" and "if the starter turns the engine and the engine does not start, then the gas supply supply is low". While heuristic rules are not always true, in dealing with an uncertain and vague world they provide a starting point for solving problems.

Expert-system rules are statements processed by software so they must have a well-defined syntax, that is, they are heuristic rules written in a form that can be processed by a computer program. The programs developed in this chapter process rules in the general form:

IF *condition-1* AND *condition-2* ... AND *condition-n* THEN DO *action-1* AND *action-2* ... AND *action-m*.

As will be seen in Section 10.1, our rules will have a LISP type syntax. Heuristic rules are often referred to as **production rules** and an expert-system shell tool is also called a production system interpreter since it works on production rules. A production system interpreter processes *if- then* rules (for example, *if* (in jeep) *then* (leave jeep)) by finding rules whose *if* clause matches a fact in the system and executing the corresponding *then* clause in the rule which adds a new fact to the system.

Facts in an expert-system are data items describing some state of the system that we are building.

An expert-system shell is a program which operates on heuristic rules and facts as data. If there are known facts that match *every* condition in a rule, then our shell program can execute the actions in this rule. Often many rules have all of their conditions satisfied by facts in a system so the shell must choose the order in which to execute the rules.

An expert-system is **forward-chaining** if it matches the conditional parts of the rules against known facts and then executes the actions. A forward-chaining production system starts at an initial state (as defined by the initial facts in the fact database) and executes rules which have all of their conditions met until either no further rules are eligible to execute or the system has reached some predefined goal state and the shell halts.

An expert-system is **backward-chaining** if it matches goals against the actions in the rules and attempts to backtrack from a goal state to the current state of a system as represented by the current facts in a system.

10.1 Simple Forward-chaining Production System Interpreter

The following example shows the implementation of a simple forward-chaining expert-system. The main limitation of this first example is the absence of matching variables; this limitation will be removed in the example in section 10.3. The first example uses a set of rules stored in the LISP variable *rules* and a set of current facts stored in the LISP variable *facts*.

Consider the following data for our first simple expert-system tool:

```
(setq rules '(
              ((in jeep) (leave jeep))
              ((leave jeep) (walk))))
(setq facts '(  (in jeep)  (has keys)))
```

The following simple program will use the fact and rule lists as data. The function *forward* is called with two arguments, a list of rules, and a list of facts. Function *forward* repeatedly matches facts to the *if* clauses in the rule list; whenever a match occurs for a fact and a rule, the *then* clause of the rule is asserted as a fact. Function *not-fact* returns a true value if the fact being tested is not in the fact database. Function *make-fact* adds new facts to the fact database.

```
;;
; File: FPS.LISP
;
; Simple forward-chaining production system interpreter (without variables)
;
;  Copyright 1990 Mark Watson.
;;

(defun forward (ruleList FactList)
  (let ((keep-going t))
    (loop
      (if (not keep-going) (return))
      (dolist (rule ruleList)
        (setq keep-going nil)
        (and
         (member (car rule) FactList :test #'equal)
         (not-fact (cadr rule) FactList)
         (progn
           (princ "Next rule to execute: ")
           (print rule)
           (terpri)
           t)
         (setq keep-going t)
         (make-fact (cadr rule) FactList))))))

(defun not-fact (testFact Facts)
  (not (member testFact Facts)))

  (defun make-fact (fact Facts)
    (setf (cdr Facts) (cons fact (cdr Facts)))
    (princ "New fact: ")
    (print fact)
    (terpri))
```

10.2 Executing the Simple Forward-chaining Production System Interpreter

The following text shows a sample execution of the simple production system interpreter listed in Section 10.1 (text in **bold** type is entered in the listener window to start the program):

```
? (forward rules facts)
Next rule to execute:
((IN JEEP) (LEAVE JEEP))
New fact:
(LEAVE JEEP)
Next rule to execute:
((LEAVE JEEP) (WALK))
New fact:
(WALK)
```

In this example execution run, function *forward* found two rules eligible to execute.

The development technique in this example is typical for writing LISP programs: start simple and incrementally build up the system. When modifying a system, start by making quick program changes that implement desired features, then rework your code to make it more readable and reusable after adding the new functionality. One of the advantages of working in a good LISP programming environment is the speed with which program changes can be implemented and tested. The time to decide whether it is worth while spending the time to make the code modules more maintainable is immediately after adding new functionality and testing.

In designing an expert-system, a good starting point is the construction of a decision tree which covers the domain (or problem area) of the proposed expert-system. In practice, systems built exclusively with decision trees tend to annoy system users by asking too many irrelevant questions. The run time performance of decision tree systems also tends to be poor. Once a decision tree has been built to cover your application area, then the heuristic rules of an expert-system can reduce both required user interaction and execution time.

10.3 Complete Forward-chaining Production System Interpreter

Our first improvement to the program in Section 10.1 will be the extension of the rule syntax to allow matching variables; for example, we want to be able to write rules such as:

```
(setq rules '(
              (if (in ?vehicle)
                 (automobile ?vehicle)
               then (leave ?vehicle))
              (if (leave ?anything)
               then (walk))))
```

Note that this new rule format offers several advantages:

- Multiple *if -then* clauses are allowed in rules.
- Variables (starting with a question mark) will match anything in the fact database.
- The "noise words" *if* and *then* make the rules more readable.

These improvements are implemented in a new function *fireRule?* which strips off the *if-then* keywords in rules and tests each *if* clause against the current fact list (second argument to *fireRule?*). The matching of *if* clauses to facts uses the new function *varMatchAll* to test for variables in the *if* clauses and sets the value of the variables to the matched values (that is, it **binds,** or sets the value of the variable to the value). For example, the *if* clause *(in ?vehicle)* would match the fact *(in jeep)* with the variable *?vehicle* bound to the value *jeep*. When new facts are asserted by a *then* clause, variable names are replaced by the values bound to them in the matching process. For example, the *then* clause *(leave ?vehicle)* would create a new fact *(leave jeep)* if the variable *?vehicle* is bound to the value *jeep*.

The following program listing implements these improvements:

```
;;
; File: FPS2.LISP
;
; Forward-chaining production system interpreter (with variables)
;
; Copyright 1990 by Mark Watson
;;

(defun forward (ruleList FactList)
 (let ((keep-going t))
  (loop
   (if (not keep-going) (return))
   (dolist (rule ruleList)
    (setq keep-going nil)
    (setq var-list nil) ;; reset current variable binding list
    (and
     (fireRule? rule FactList var-list)
     (progn
      (princ "From rule:")
      (print rule)
      (terpri)
      t)
    (setq keep-going t))))))
```

```
;;
; Check to see if a new fact has already been added to the fact database:
;;

(defun not-fact (testFact Facts)   (not (member testFact Facts :test #'equal)))

;;
; Add a new fact to the fact database. Note that this function
; alters the value of the second argument with setf.
;;

(defun make-fact (fact Facts)
  (if fact ;; make sure that fact is not just nil
    (let ()
      (setf (cdr Facts) (cons fact (cdr Facts)))
      (princ "New fact: ")
      (print fact)
      (terpri))))

;;
; Functions for variable matching:
;;

(defun isVar? (x)  ;; check to see if the first character in name is a ?
  (and (symbolp x) (char-equal (char (symbol-name x) 0) #\?)))

(defun varMatch (v-list fact var-list) ;; wild-card pattern match
  ;; Make sure the lists are the same length:
  (if (equal (length v-list) (length fact))
    (let ((match-ok t))
      (dolist (v v-list)
        (if (not (equal v (car fact))) ;; no match, but check for a variable:
          (if (isVar? v)
            (setq var-list (cons (list v (car fact)) var-list)
                  fact (cdr fact))
            (return (setq match-ok nil))) ;; no match and no variable
          (setq fact (cdr fact))))
      (if match-ok
        (if var-list ;; we need to return a non-nil value so return the
          var-list   ;; new updated variable list
          T)))))     ;; or return T to signal a match to the calling function.

(defun varMatchAll (v-list theFacts var-list &aux ret)
  (dolist (fact theFacts)
    (setq ret (varMatch v-list fact var-list))
    (if ret
      (if (not (equal ret var-list))
        (return ret)
        ret))))
```

```
;;
; Test for valid rule-firing. Return updated variable binding list
; if the rule is eligible to execute:
;;

(defun fireRule? (rule FactList var-list)
 (if (not (equal (car rule) 'if)) ;; simple check on rule syntax
   (let ()
    (princ "Illegal rule:")
    (print rule))
   (let ((valid t) added-fact)
    (setq rule (cdr rule))
    (do ((check (car rule))) ; check = if clauses
       ((or
         (null rule)
         (equal check 'then)))
       (let (new-var-bindings)
         (setq new-var-bindings
              (varMatchAll check FactList var-list))
         (if new-var-bindings
           (if (not (equal new-var-bindings T))
             (setq var-list (append new-var-bindings var-list)))
           (return (setq valid nil)))) ;; if clause that does not match
       (setq check (car rule)))
    (if (equal (car rule) 'then)
      (if valid ;; execute the then clauses of the rule:
       (let (new-fact)
         (setq rule (cdr rule))
         (dolist (thenClause rule)
           (setq new-fact (substituteVariables thenClause var-list))
           (if (not-fact new-fact FactList)
             (let ()
                 ;; Surgically insert new-fact
               (make-fact new-fact FactList)
               (setq added-fact t))))
         added-fact))))))

;;
; Use the variable binding list to substitute values for variables:
;;
(defun substituteVariables (fact var-list)
 (dolist (var var-list)
   (let ((variable-name (car var))
        (bound-value (cadr var)))
     (setq fact
          (mapcar
           #'(lambda (x)
              (if (equal x variable-name)
                bound-value
              x))
          fact))))
 fact)

;;
; Test data:
;;

(setq rules '(
          (if (in jeep) (has ?object) then (drive jeep))
          (if (has ?x) then (drop ?x))
          (if (drive jeep) then (be-at store))
          (if (in jeep) then (leave jeep))
          (if (leave jeep) then (walk))))

(setq facts '(  (in jeep)   (has keys)))

;;
; Test execution of the system automatically as this file is loaded:
;;

(forward rules facts)
```

10.4 Executing the Complete Forward-chaining Production System Interpreter

The following listing shows the output from the example program listed in section 10.3. Notice that the rules used in the example contain variables starting with a question mark. In the following text listing, text with bold type was typed as input to the production system interpreter and text in a normal typeface was output from the program:

```
? (setq rules '(
          (if (in jeep) (has ?object) then (drive jeep))
          (if (has ?x) then (drop ?x))
          (if (drive jeep) then (be-at store))
          (if (in jeep) then (leave jeep))
          (if (leave jeep) then (walk))))
((IF (IN JEEP) (HAS ?OBJECT) THEN (DRIVE JEEP))
 (IF (HAS ?X) THEN (DROP ?X))
 (IF(DRIVE JEEP) THEN (BE-AT STORE))
 (IF (IN JEEP) THEN (LEAVE JEEP))
 (IF (LEAVE JEEP) THEN (WALK)))
? (setq facts '(
           (in jeep)
           (has keys)))
```

```
((IN JEEP) (HAS KEYS))
? (forward rules facts)
New fact:
(DRIVE JEEP)
From rule:
(IF (IN JEEP) (HAS ?OBJECT) THEN (DRIVE JEEP))
New fact:
(DROP KEYS)
From rule:
(IF (HAS ?X) THEN (DROP ?X))
New fact:
(BE-AT STORE)
From rule:
(IF (DRIVE JEEP) THEN (BE-AT STORE))
New fact:
(LEAVE JEEP)
From rule:
(IF (IN JEEP) THEN (LEAVE JEEP))
New fact:
(WALK)
From rule:
(IF (LEAVE JEEP) THEN (WALK))
NIL
?
```

10.5 Backward-chaining Expert-systems

Both forward and backward-chaining expert systems solve search problems. The next example program demonstrates the techniques for writing recursive backwards chaining search programs, that is, systems which start with one or more goal states and apply the rule database to attempt to work backwards to the present state of the system (as encoded in the fact database). The next example program uses the same rule and fact database format as the first example in this chapter; in addition, a target goal state is specified. For example, the initial state of the system could be specified with the following data:

```
(setq rules '(
              ((in jeep) (leave jeep))
              ((leave jeep) (walk))))

(setq facts '( (in jeep) ))

(setq goal '(walk))
```

The first example in this chapter started with the fact *(in jeep)* and used the above rules to produce the following facts: *(leave jeep)* and *(walk)*. The next example program starts with a goal state (walk) and one initial fact (in jeep) and forms a plan by searching backwards from the goal state until a **path** is found to the initial state of the system. Here a path is defined as an ordered set of rule executions which produce the initial state of the system (that is the initial facts) starting at the goal state. The ordered set of rule executions that transitions the system from the goal state to the initial state of the system is called an **inference chain.** The first rule *if (in jeep) then do (leave jeep)* operating on the state: *(leave jeep)* allows the **transition** backwards from the state: *(leave jeep)* to *(in jeep)* to occur. The program is a fairly simple backward-chaining production system interpreter; the purpose of this example program is to provide an example search program which enumerates all possible **transitions** from a given state and **recursively** calls itself for each possible transition until either a goal state is reached or no further transitions remain to be searched. In Chapter 11 we will look at a similar problem: heuristically finding cost effective paths through networks using heuristic search. This chapter is both a gentle introduction to a complex heuristic search program presented in Chapter 11 and a reinforcement of the principle of using heuristic rules to limit search space and time.

10.6 Backward-chaining Production System Interpreter

The following example program starts at the goal state and recursively applies the rule database until either the initial state is reached or no more rules can be chained backwards. The function *backward* is called with a list of rules, a list of facts, a desired goal state, and a list of current deductions. The list of deductions is *nil* the first time that function *backward* is called; *backward* calls itself recursively adding to the deduction and fact lists until the goal state is contained in the fact list. The first two *if* tests in function *backward* are simply for debug printout. The first statement in function *backward* which performs work is a test to see if the current goal is already in *FactList* in which case the function simply returns the value of its current inference chain. If the current goal is not in the current fact list, the function loops over all rules in the variable *ruleList* first checking whether a rule with its *then* clause, the *(cadr of-the-rule),* is equal to the current goal and if its *if* clause is in the fact list, in which case the search is complete. In the general case, function *backward* will now recursively call itself with a new goal: the *if* clause of each rule in the rule list.

```
;;
; File: BPS.Lisp
;
; Simple backward-chaining production system interpreter
;   Copyright 1990 by Mark Watson
;;

(setq *BT-DEBUG* T)  ;; flag for debug output

(defun backward (ruleList FactList Goal inferenceChain)
  ;;
  ; Test for desired debug printout:
  ;;
  (if *BT-DEBUG* (terpri))
  (if *BT-DEBUG*
    (pprint (list "Entering backward: ruleList=" ruleList "FactList=" FactList
             "Goal=" Goal "inferenceChain=" inferenceChain)))
  ;;
  ; Test having already found the path from the goal
  ; back to the initial state of the system:
  ;;
  (if (member Goal FactList :test #'equal)
    inferenceChain ;; done with search
    ;; Find a rule with the then clause equal to the goal
    ;; and backtrack trying to prove the if clause in the rule:
    (dolist (rule ruleList)
      (if (equal (cadr rule) Goal)   ;; (cadr rule) equals the then clause in the rule
        (if (member (car rule) FactList :test #'equal)
          (pprint (list "Solution to backtracking problem:"
                  (append (append FactList  (list Goal))
                          inferenceChain)))
          ;;
          ; Recursively call this function backward passing the if clause
          ; of the current rule being processed as the goal to search
          ; backwards from and append the current goal to the inference chain:
          ;;
          (let ((search (backward ruleList FactList (car rule)
                         (cons Goal inferenceChain))))
            (if search
              (return (cons (list rule FactList) search)))))))))

;;
;Test data. This example rule set will automatically execute when
; this sample file is loaded into your Common Lisp environment:
;;
```

```
(setq rules '(
             ((A) (B))    ;; if A then B
             ((B) (C))    ;; if B then C
             ((C) (X))    ;; if C then X–not used in inference chain
             ((Z) (D))    ;; if Z then D–not used in inference chain
             ((C) (D))    ;; if C then D
             ((D) (E))    ;; if D then E
             ((E) (F))))  ;; if E then F

(setq facts '((A)))
(setq goal '(F))

;;
; Run the system:
;;

(setq *BT-DEBUG* nil)  ;; turn off debug output

(progn
  (terpri) (terpri) (princ "********************")
  (terpri)
  (terpri)
  (backward rules facts goal nil))
```

10.7 Executing the Backward-chaining Production System Interpreter

The sample data at the end of this example has two rules that are not used in the backward-chaining inference chain for a test: note in the following output from the program that the inferences *(X) to (C)* and *(D) to (Z)* are never made since the search for these two inferences does not lead back to the initial state of the system. The following output was captured while executing this example two times: once without debug output and one with the debug flag **BT-DEBUG** set to a non-nil (true) value. Text in a bold typeface was typed into the Lisp listener window to initialize data for the sample program and then to execute it. Text in a normal typeface is program output.

```
? (setq rules '(
           ((A) (B))   ;; if A then B
           ((B) (C))   ;; if B then C
           ((C) (X))   ;; if C then X
           ((Z) (D))   ;; if Z then D
           ((C) (D))   ;; if C then D
```

```
          ((D) (E))   ;; if D then E
          ((E) (F)))
(((A) (B)) ((B) (C)) ((C) (X)) ((Z) (D)) ((C) (D)) ((D) (E)) ((E) (F)))

? (setq facts '((A)))
((A))
? (setq goal '(F))
(F)
? (setq *BT-DEBUG* nil)
NIL
? (backward rules facts goal nil)

("Solution to backtracking problem:"
 ((A) (B) (C) (D) (E) (F)))
NIL
? (setq *BT-DEBUG* T)
T
? (backward rules facts goal nil)


("Entering backward: ruleList="
 (((A) (B)) ((B) (C)) ((C) (X)) ((Z) (D)) ((C) (D)) ((D) (E)) ((E) (F)))
 "FactList=" ((A))
 "Goal=" (F)
 "inferenceChain=" NIL)

("Entering backward: ruleList="
(((A) (B)) ((B) (C)) ((C) (X)) ((Z) (D)) ((C) (D)) ((D) (E)) ((E) (F)))
 "FactList=" ((A))
 "Goal=" (E)
 "inferenceChain=" ((F)))

("Entering backward: ruleList="
(((A) (B)) ((B) (C)) ((C) (X)) ((Z) (D)) ((C) (D)) ((D) (E)) ((E) (F)))
 "FactList=" ((A))
 "Goal=" (D)
 "inferenceChain=" ((E) (F)))

("Entering backward: ruleList="
(((A) (B)) ((B) (C)) ((C) (X)) ((Z) (D)) ((C) (D)) ((D) (E)) ((E) (F)))
 "FactList=" ((A))
 "Goal=" (Z)
 "inferenceChain=" ((D) (E) (F)))

("Entering backward: ruleList="
(((A) (B)) ((B) (C)) ((C) (X)) ((Z) (D)) ((C) (D)) ((D) (E)) ((E) (F)))
 "FactList=" ((A))
 "Goal=" (C)
 "inferenceChain=" ((D) (E) (F)))

("Entering backward: ruleList="
(((A) (B)) ((B) (C)) ((C) (X)) ((Z) (D)) ((C) (D)) ((D) (E)) ((E) (F)))
 "FactList=" ((A))
 "Goal=" (B)
 "inferenceChain=" ((C) (D) (E) (F)))
("Solution to backtracking problem:"
 ((A) (B) (C) (D) (E) (F)))
NIL
```

10.8 Public Domain and Commercial Expert-system Shells

The examples in this chapter were designed to provide insight into the operation of production-system interpreters. For serious development of expert-systems, it is better to use a standard tool that is integrated with the programming environment used for your application development. Because most expert-systems are relatively small programs embedded in much larger system, integration with the development language and utilities is very important. It is also important for the shell language to provide a rich set of utilities for performing partial pattern matching on data elements.

An excellent tool for developing Common LISP applications is the public domain OPS5™ language written at Carnegie-Mellon University by Charles Forgy. OPS5 is a forward-chaining system like the first two example programs in this chapter and its source code is available, giving the system developer full control over the production system interpreter.

If C is the language for project development, the public domain CLIPS™ system is a good choice. CLIPS was developed by NASA for in house use and is distributed for a copying charge by the US government COSMIC™ software clearing house. CLIPS is distributed as C source code. An excellent commercial shell written in C is Nexpert™ from Neuron Data Corporation.

10.9 Projects

1. All *then* actions in both forward-chaining programs in this chapter add a new fact to the system. Modify the complete forward-chaining program by adding the new action keyword *remove* which removes a specified fact from the fact database. Modify the system to allow rules such as:

```
(setq rules '(
  (if (location town) (money ?amount)
     then   (remove 2))
))
```

Note that *remove* takes one argument which refers to the pattern number in the *if* part of the rule. In this case *(remove 2)* indicates that the fact in the fact database which matches the second *if* pattern *(money ?amount)* in the rule should be removed.

2. The complete forwards-chaining program performs limited syntax-checking of the rules (check the rule for the *if* noise word). Add the following syntax checks to function *fileRule?*:

   - Check for the existence of the *then* noise word.
   - Insure that there is at least one *if* test in the rule.
   - Insure that there is at least one *then* action in the rule.

   Note that rule-syntax checking is repeated each time the rule database is checked. Remove syntax-checking from function *fireRule?* and perform syntax checking of all rules once in function *feedForward.*

3. Modify the forwards-chaining program to add a new *then* action: *(halt).* This new action causes the production system interpreter (the function *feedForward*) to immediately halt even if further rules are eligible to execute.

4. Modify the forwards-chaining program to add an optional rule priority to the allowed syntax for a rule.
   Hint: if several rules are eligible to execute, execute those with the highest priority first. Also, simply sort the initial rule database in decreasing order of priority, and modify *feedForward* to start searching for a valid rule at the beginning of the rule list whenever a rule fires. Why is this second step necessary?

5. Modify the backwards-chaining program to accept multiple *if* statements in rules.
   Hint: at the end of function *backward,* call *backward* recursively for each *if* clause, but immediately return false (*nil*) if the recursive call to *backward* ever returns false.

6. Modify the backwards-chaining program to allow variables in the rules using the same format as the second example program in chapter 10.
   Hint: you will need to build variable bindings of the form: ((?var-1 value-1) (?var-2 value-2) and (?var-n value-n)) and pass these bindings to *backward* when *backward* is making recursive calls to itself.

11
Heuristic Network Search Algorithm

Many AI applications require a search for a **best** (or shortest) path through a network. Applications for network searches include routing problems (such as aircraft or sales route planning and terrain traversal for robotic vehicles).

Three popular search strategies for finding a short path through a network are: depth-first, breadth-first, and heuristically directed breadth-first. Depth-first searches are very inefficient, in general, because the first available path is searched exhaustively, even if it turns out to be a very long path. The advantages of depth-first searches are the ease of implementation and small data storage requirements. Breadth-first searches are much better since the search space tends to "fan out" from the starting node until the goal node is located. Breadth-first searches have a much larger run-time memory requirement than depth-first searches. The heuristically directed breadth-first search attempts to search possible paths in the order of paths that are heuristically judged to have the highest probability of being the best path. This directed search causes the search space to "fan out" preferentially towards the goal node. This directed search is very simple to implement since we have an accurate cost function for ordering the possible paths for each node being searched: the cost of any possible path is taken to be the Cartesian distance from the first node on a possible path to the goal node. This chapter will present a good technique for performing an efficient heuristic breadth-first network search, the A* search (Hart, Nilsson, and Raphael, 1968).

11.1 Theory of the A* Search

We will discuss the theory of A* search in the framework of a practical application: calculating the shortest path between cities that are connected by roads. The data that the A* algorithm uses is a set of **nodes** (the cities in this example), a set of **edges** or **paths** between nodes (the roads), a starting node (or city) and a goal node (the city we want to reach).

The computational steps used in the in the A* algorithm are simple to describe. At the starting node, we build a list of all nodes that are directly connected to the starting node by a path. We sort this list of nodes in order of proximity to the goal node (list the cities closest to the target city first). We then start a breadth-first search of all nodes in this list; at each node we search from, we sort the list of connected nodes by proximity to the goal node. The search terminates as soon as we search a node directly connected to the goal node.

This last step of sorting possible path segments by proximity to the goal node, in combination with **hashing** (storing computed information with an index key to find it quickly) the calculated distance between directly or indirectly connected nodes makes a single search through the network much faster. The hashing of precomputed distances greatly speeds up subsequent searches through the network. Our example makes use of both of these tricks.

11.2 A* Network Search Program

The function *A*search* is called with the names of the starting and goal nodes for the search. Function *A*search* uses the local list *possible-paths* to cache the Cartesian distances between nodes in the network as they are calculated. The helper function *a*-helper* performs a breadth-first search by starting with the first possible path (this list is kept sorted so that the possible paths with an ending node closest to the goal node are kept first on the list) and fanning the search out preferentially towards the goal node. Note that this search stops when the first path segment on the possible path list contains the goal node.

The function *init-network* precalculates the lengths of all path segments in the network. The function *init-path-list* puts calculated path lists as a property value on each node. By caching precalculated path costs for each node, the example program avoids recalculating any path sequence. For example, if after calculating the path between nodes *n1* and *n11* in Figure 11.1, function *A*search* is then asked to calculate the best path between nodes *n2* and *n11*, no real search is necessary since the path sequence from *n2* to *n11* will already be cached at node *n2*.

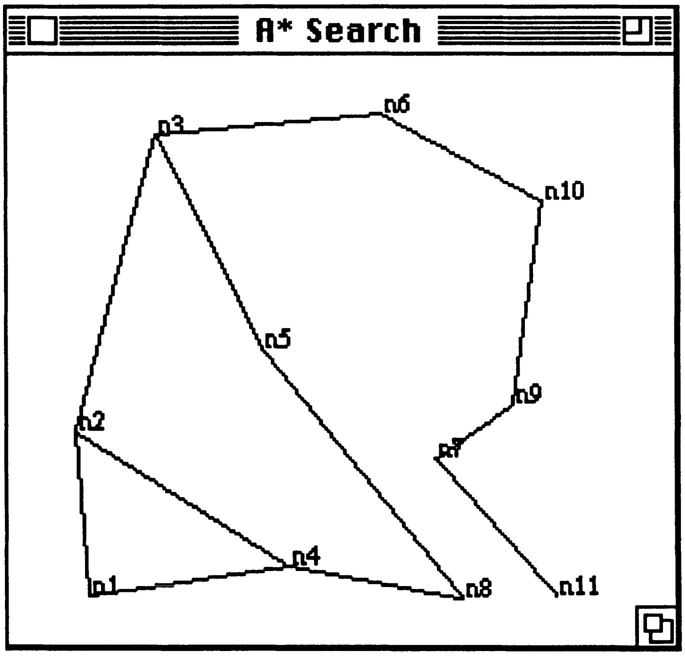


Figure 11.1. Sample Network with 11 Nodes and 12 Edges

The function *enumerate-children* collects all path segments from a specified starting node to the goal node. If a stored path is found cached on the starting node, function *consider-best-path* uses the stored information; if a stored path is not found cached in the starting node, function *consider-all-nodes* is called to search for a best path.

The function *Collect-ascending-search-list-order* places the possible paths fanning out from a node in a "best-first" order using previously calculated direct distances ("as the crow flies") from each node to the goal node.

The function *test1* in the sample program demonstrates the initialization of data for the A* search program. Two variables define the network used by the example program: *nodes* and *paths*. In *test1*, eleven test nodes are created as global data items with values of Cartesian coordinates. For example, node *n1* is defined to be at the location specified by the coordinates (30, 201) in some arbitrary coordinate system. The global variable *paths* is set to a list of node pairs defining the connections in the network.

Function *test1* calls function *init-network* to calculate and cache the path lengths of all path segments in the network. Function *test1* then plots the nodes in a graphics window (assuming the graphics library from chapter 2 has been loaded).

Function *test2* calls the function *A*search* which returns a path list for the best route from node *n1* to node *n11*. The best path is plotted in a graphics window.

```
;;
; File A*.lisp
;
; Perform a heuristic A* search between the start and goal nodes:
;
; Copyright 1990 by Mark Watson
;;

(defun A*search (start goal &aux possible-paths best)
 (setq possible-paths
      (list
       (list
        (dist-between-points
          (eval start)
          (eval goal))
          0
          (list start))))
 (defun a*-helper () ;; Note: this function is nested inside of function A*
  (if possible-paths ;;        so its lexical scoping makes it invisible outside A*
    (let ()
      (setq best (car possible-paths))
      (setq possible-paths (cdr possible-paths))
      (if (equal (first (caddr best)) goal)
        best
        (let ()
          (dolist (child (enumerate-children best goal))
            (setq possible-paths
                 (Collect-ascending-search-list-order
                   child possible-paths)))
          (a*-helper))))))
  (a*-helper))

;;
 ; Define memonics to fetch x,y coordinates from nodes:
 ;;

(defun Y-coord (x) (truncate (cadr x)))
(defun X-coord (x) (truncate (car x)))

;;
 ; Initialize the network by calculating path lengths
 ; of all path segments and caching this information
 ; at each node for future use:
 ;;

(defun init-network ()
  (setq paths (init-lengths paths))
  (init-path-list nodes paths))

;;
 ; Add the length of the path to each path in the path list:
 ;;

(defun init-lengths (pathlist)
  (let (new-path-list pathlength path-with-length)
    (dolist (path pathlist)
      (setq pathlength (slow-path-length path))
      (setq path-with-length (append path (list pathlength)))
      (setq new-path-list (cons path-with-length new-path-list)))
    new-path-list))


;;
 ; Place the path-list property on each node :
 ;;

(defun init-path-list (nodes paths)
  (dolist (node nodes)
        (put node 'path-list
            ;; let returns all paths connected to node:
```

```
        (let (path-list)
         (dolist (path paths)
          (if (equal node (start-node-name path))
           (setq path-list
                (cons (list (end-node-name path)
                            (path-length path))
                      path-list))
           (if (equal node (end-node-name path))
            (setq path-list (cons (list (start-node-name path)
                                        (path-length path))
                                  path-list)))))
         path-list ))))

;;
; "As the crow flies" distance between
;  the starting and ending nodes on a path:
;;

(defun slow-path-length (path)
  (dist-between-points (start-node path) (end-node path)))


;;
; Define mnemonic for getting path-length from a node:
;;

(defun path-length (x) (caddr x))

;;
; Calculate the Cartesian distance between points:
;;

(defun dist-between-points (point1 point2)
  (let ((x-dif (- (X-coord point2) (X-coord point1)))
        (y-dif (- (Y-coord point2) (Y-coord point1))))
  (sqrt (+ (* x-dif x-dif)  (* y-dif y-dif)))))

;;
; Define memonics for accessing data in lists:
;;

(defun start-node (path) (eval (car path)))
(defun end-node (path) (eval (cadr path)))
(defun start-node-name (x) (car x))
(defun end-node-name (x) (cadr x))
```

```
(defun first-on-path (x) (caddr x))
(defun goal-node (x) (car x))
(defun distance-to-that-node (x) (cadr x))

;;
 ; Generate all child paths from a node to the goal:
 ;;

(defun enumerate-children (node goal)
  (let* (
          (start-to-lead-node-dist (cadr node)) ;; distance already calculated
           (path (caddr node))
           (lead-node (car path)))
    (if (get-stored-path lead-node goal)
      (consider-best-path lead-node goal path start-to-lead-node-dist)
      (consider-all-nodes lead-node goal path start-to-lead-node-dist))))

;;
 ; Set "best-first" search towards next node:
 ;;

(defun consider-best-path (lead-node goal path distance-to-here)
  (let* (
            (stored-path (get-stored-path lead-node goal))
            (stored-dist (distance-to-that-node stored-path))
            (first-node (get-first-node-in-path lead-node goal))
            (dist-to-first (+ distance-to-here
                              (get-stored-dist lead-node first-node)))
            (total-estimate (+ distance-to-here
                               (get-stored-dist lead-node goal)))
            (new-path (cons first-node path)))
  (list (list total-estimate dist-to-first new-path))))

;;
 ; Return optimum path between two nodes (if available):
 ;;

(defun get-stored-path (start goal)
  (if (equal start goal)
      (list start 0)
      (assoc goal (get start 'path-list))))
```

```
;;
; If optimum path between nodes has not been calculated
; already, consider all nodes connected to lead-node:
;;

(defun consider-all-nodes (lead-node goal path start-to-lead-node-dist)
 (let (dist-to-first total-estimate new-path new-nodes)
  (dolist (node (collect-linked-nodes lead-node))
   (if (node-not-in-path node path)
    (let ()
     (setq dist-to-first (+ start-to-lead-node-dist
                           (get-stored-dist lead-node node)))
     (setq total-estimate (+ dist-to-first
                            (dist-between-points
                             (eval node)
                             (eval  goal))))
     (setq new-path (cons node path))
     (setq new-nodes (cons (list total-estimate
                               dist-to-first
                               new-path)
                          new-nodes)))))
  new-nodes))


;;
; Collect all nodes connected to node:
;;

(defun collect-linked-nodes (node)
 (let (links)
  (dolist (link (get node 'path-list))
   (if (null (first-on-path link))
    (setq links (cons (goal-node link) links))))
  links))

;;
; Check to see if a node is on a given path:
;;

(defun node-not-in-path (node path)
 (if (null path)
  t
  (if (equal node (car path))
   nil
   (node-not-in-path node (cdr path)))))
```

```
;;
; Retrieve the distance between nodes:
;;

(defun get-stored-dist (node1 node2)
  (distance-to-that-node (get-stored-path node1 node2)))

;;
; Order the paths leading out of a node in "best-first" order:
;;

(defun Collect-ascending-search-list-order (a l)
  (if (null l)
    (list a)
    (if (< (car a) (caar l))
      (cons a l)
      (cons (car l) (Collect-ascending-search-list-order a (cdr l))))))


;;
; Return the first node in a path towards the goal node:
;;

(defun get-first-node-in-path (start goal)
  (let (first-node)
    (setq first-node  (first-on-path (get-stored-path start goal)))
    (if first-node first-node goal)))

;;
;     Throw away test code:   Execute (test1) then (test2) to display
;     a network and performs an A* search between two nodes:
;;

(defun test1 ()
  (setq
      n1 '(30 201)
      n2 '(25 140)
      n3 '(55 30)
      n4 '(105 190)
      n5 '(95 110)
      n6 '(140 22)
      n7 '(160 150)
      n8 '(170 202)
      n9 '(189 130)
      n10 '(200 55)
      n11 '(205 201))
  (setq nodes
      '(n1 n2 n3 n4 n5 n6 n7 n8 n9 n10 n11))
  (setq paths
      '((n1 n2) (n2 n3) (n3 n5) (n3 n6) (n6 n10)
        (n9 n10) (n7 n9) (n1 n4) (n4 n2) (n5 n8)
        (n8 n4) (n7 n11)))
  (init-network)
  (init-plot)
  (dolist (p paths)
    (let ((from (eval (car p)))
          (to (eval (cadr p))))
      (plot-line
        (x-coord from)
        (y-coord from)
        (x-coord to)
        (y-coord to))))
  (dolist (n nodes)
    (let ((n-val (eval n)))
      (plot-string
        (+ 2 (x-coord n-val))
        (y-coord n-val)
        (princ-to-string n)))))

(defun test2 ()
  (defun draw-path (pl)
    (print pl)
    (pen-width 3)
    (let ((node1 (car pl)))
      (setq pl (cdr pl))
      (dolist (p pl)
        (plot-line (x-coord (eval node1)) (y-coord (eval node1))
                   (x-coord (eval p)) (y-coord (eval p)))
        (setq node1 p))))
  (draw-path (caddr (A*search 'n1 'n11))))
```

11.3 Executing the A* Search Program

Executing function *test1* generates the plot seen in Figure 11.1 showing 11 nodes in a sparsely connected network. The graphics library from chapter 2 must be loaded before running function *test1*.

Executing function *test2* in the sample program calculates the best path between nodes N1 and N11 and hilights the path in the graphics display as seen in Figure 11.2.

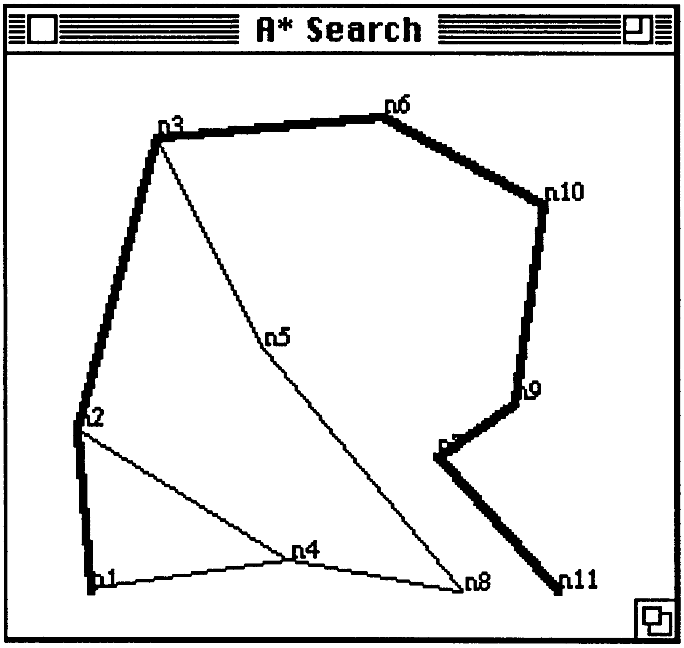


Figure 11.2. The Best Path between Nodes n1 and n11

The program presented in this chapter can search a network with a 100 nodes for a path in a few seconds on a Mac II running Macintosh Common LISP. Notice that in our example, the heuristically directed breadth-first search algorithm is "misled" rather badly since this cost function will cause the search to proceed first to node N4 and then to node N8; the program still finds the correct shortest path, but the search is more expensive computationally.

11.4 Projects

1. Write a program to calculate best paths between cities in the USA. Superimpose a Cartesian grid over a map with units in miles. Then enter the location of each major city as a point (x, y). Enter a list of roads where each road is defined as a city pair. Using this data, plot the cities in the graphics window. Your program should accept two cities as input (use the mouse pointing device, or require typing in the names) and plot out the best path between the cities using the available roads. Reuse the test functions *test1* and *test2* , changing the data initialization code.

2. Modify the example program to allow each path to have a speed-cost attribute. The cost function *dist-between-points* in the program in section 11.2 uses the

Cartesian distance between two nodes as the cost for traveling on a path directly connecting the nodes. Modify the cost function to calculate the cost as path length divided by the speed attribute. The cost function would then be time-to-travel, not distance.

3. Simulate the traversal of a robot vehicle over rough terrain. Use the program modifications from the second project in this section. Create a two dimensional array that has 10x10 grid cells and fill each grid cell with a random value between 1 and 5; this random value will be a speed-cost attribute for moving through the grid cell. Allow movement of the vehicle to be between adjacent grid cells only (not between diagonally connected cells). Set up your simulation to plot the grid cells with gray-scale values (indicating the speed at which the vehicle can move through a grid cell) and the calculated best path from the upper left to the lower right grid cell. Every time your simulation program executes, fill the grid cells with new random speed-cost attributes. Set your simulation program up to continuously loop over new terrains.

12
A Chess-Playing Program

Computer chess programs have long been a testing ground for developing search techniques. Chess programs use some form of **lookahead** (prediction of future moves) search combined with **static evaluation** (judging the relative worth of a chess position without lookahead, that is by counting material worth of both black and white pieces and comparing the relative mobility of those pieces). We will develop a chess playing program in this chapter which relies mostly on static evaluation using heuristic chess knowledge. The program will usually give the impression of strong positional play although it plays a poor tactical game. The advantages of this heuristic approach (as compared to brute force search) are: faster execution in a LISP environment, easier to add and test new chess heuristics, and a simpler program. The static evaluation function, in calculating which pieces are subject to capture, effectively performs a two-**ply** (that is, half moves, by either the white or black pieces) lookahead search.

12.1 Computer Chess History

Claude Shannon described two search techniques that he called **type A** and **type B** (Shannon 1950). The type A search technique looked at all possible moves from a given position. For example, there are usually about 40 possible (legal) chess moves from a typical board position; to look three moves ahead (two moves for each player) requires the examination of about $40^{(3*2)}$, or 40^6 (4,096,000,000) board positions. Because of the impracticality of deep searches using the type A search strategy, Shannon predicted that high-performance game-playing programs would instead use the human-style type B strategy in which the program would not examine all possibilities, but use heuristic rules derived from consultation with human chess experts. In this way a program would be able to examine only "good" moves, without, it was hoped, overlooking many strong moves. If a chess program examines only six plausible moves per turn, then a three-move lookahead requires the examination of $6^{(3*2)}$, or 6^6 (46656) board positions.

While he was a student at MIT, Richard Greenblatt, a world class LISP programmer, wrote an excellent chess-playing program (which used to clobber me when I was learning to play chess) that used Shannon's type B strategy. However, Greenblatt's program and all other type B search strategy chess programs never considered, and thus missed, many good moves.

Human chess masters are capable of very fine evaluation of static board position (judging the relative value of a chess position without lookahead by counting both material worth of both black and white pieces on the board and the relative mobility and usefulness of the pieces) because a given board position triggers many associative memory recalls (as mentioned in chapter 3) based on the study of thousands of games with many thousands of unique board positions. Hardware neural network chips will certainly be useful in refining static evaluation functions in the future. Unfortunately, software static evaluation functions for complex games like chess and Go are not adequate by themselves; they must be augmented by deep lookahead.

Lookahead is accomplished by using a **min-max** search where each side is assumed to choose its moves to minimize or maximize the static evaluation of the board position at the end of a lookahead search. All legal moves are generated for the side to move. These moves are tried one at a time while all the possible moves for the other side are generated. As discussed in chapter 11, there are two general types of search strategies: depth-first and breadth-first. Chess programs usually use a depth-first search to eliminate the extra storage requirements for implementing a breadth-first search. The **depth** of the min-max search is the number of plies (that is, half moves, of either the white or black pieces) searched.

The heart of any game-playing program using a min-max lookahead is the static evaluation function, which given any board position will return a numeric rating of how good the board position is for each player (the value for player A is equal to minus the value for player B).

Chess programs that use deep lookahead require an inexpensive static evaluation function since it must be applied to every board position occurring in the search and superior chess-playing programs use the simpler Shannon type A search strategy exclusively. However, the static evaluation function is still very important for two reasons. The first reason is obvious: even though the program generates all possible board positions looking several moves into the future, the ultimate move choice is calculated by the application of the static evaluation function to each **terminal** board position (that is, to those board positions for which there is no further searching). The second reason the evaluation function is so important is that the min-max search method makes use of it when performing **alpha-beta pruning**, that is, by ordering the search of each possible move from a given position to search the best moves first. Frequently, a refutation move is found early which allows the elimination of the search of large segments of the search tree, without affecting the calculated move choice. The percentage of the potential move tree pruned using alpha-beta cutoffs is in general a function of how good the static evaluator is. The chess-playing program developed in section 12.3 uses an expensive but very accurate static evaluation function. This has the disadvantage of drastically reducing the number of board positions that can be searched in a reasonable amount of time but the advantage of the quality of the chess moves selected from a good positional evaluation. Positional considerations include mobility of chess-pieces, control of the center and control of the squares around the kings.

12.2 Design of the Example Chess Program

Commercial chess-playing programs are certainly not written in LISP for performance reasons. Even with a good compiler, a chess-playing program written in LISP will play much slower than one coded in assembler language. However, in 1978, I enjoyed working on a chess-playing program written in interpreted Basic on an Apple II computer even though the program took about ten minutes per move. I now use a Macintosh running Macintosh Common LISP which runs about 200 times faster (about 2 seconds per move) than interpreted Basic on the Apple II. Combined with a good LISP compiler and a fast microcomputer, the example program in section 12.3 should provide hours of enjoyment since it is very easy to modify and experiment with.

The design of the program is fairly simple in that it has just three main modules: a static evaluation function, a move generator, and board database update module which uses the first two modules to calculate aand make a move.

The static evaluation function works by calculating a **static evaluation value** (zero if the white and black sides are equal, positive for a superior black position, and negative for a superior white position). The following parameters are used in this calculation:

- Material (value of pieces left on the board)
- Number of times pieces are guarded
- Mobility of playing pieces
- Control of central squares by pawns
- Control of central squares by pieces
- Control of squares adjacent to both kings
- Attack on pieces by less valuable pieces

The function *value* implements the calculation of the static evaluation value for a given board position.

The user interface for the chess-playing program is a simple read loop with a text display of the board. For the sake of simplicity, the program only plays the black pieces. It is left as a simple exercise in section 12.5 to modify the example program to play either side.

12.3 Example Chess Program

This program is only about five pages of code (not including comments): it has been designed and implemented with the objective of being short and easy to understand and modify. The chess program uses a clever indexing scheme for board squares to make move generation easier. The comments in the second page of the program listing show the mapping between this indexing scheme and the squares on the board. If a piece on square number *N* is to move forward one square, its new square index can be calculated by adding 10 to the old value. Rook moves can thus be calculated by adding or subtracting either 10 or 1 to the current square value until the rook attempts to move to a non-empty square. Bishop moves can also be calculated in a simple loop by adding the increment values -9, -11, 9, or 11 to the square index. The increment values for a given piece type are calculated in function *goto* by setting *pieceType* to the absolute value of the board value and using *pieceType* as an index into the list **index**, which is used as an index into the move increment list **piece**. As a simple example, suppose that function *goto* is called with a square index of 66: the value of (aref **board** 66) equals 3 (Bishop), which is used as an index into the **index** list. The value of *(nth 3 *index*)* equals 10, which is an index into the piece movement increment counter list **piece**. Legal piece increments for bishops are the numbers starting at index 10 in list **piece** and ending with an entry of zero in list **piece**. The data declarations and initialization for these lists and arrays are in the first 15 lines of the chess program.

The chess program uses a global variable **board** to hold the current board position. For a given board position, the computer keeps track of the number of times both sides control each square on the board; the counter array **bcon** is used to count the number of times the black (computer) pieces control each square on the board; the array **wcon** is used for the corresponding white (human opponent) pieces.

The function *initChess* sets flags indicating that neither king has moved, sets the move number counter **move-num** to zero, and initializes the **board** array.

The functions *printPiece* and *printBoard* are used to print out the current board position. When printing the value of a piece, if the value of the piece is zero, then an empty square is printed as either a *X* or a period (.) to represent black and white squares on the chessboard.

The function *posib* calculates a list of all possible moves on the board. The utility function *goto* is used to calculate all possible moves by a piece on a specified square of the chessboard.

The function *goto* returns a list of squares that a piece on a desired square number can move to. Most of the complexity of calculating piece moves in function *goto* is in the code for pawn moves since pawns' behavior depends on their square number (pawns on their original squares can move one or two squares forward and if they are capturing then make diagonal moves).

The function *value* returns a static evaluation using the seven heuristic rules listed in section 12.2. Function *value* is passed the square number that is being moved to.

The function *board-pr* is a utility function printing the name of a square index. For example, square index 22 would be converted to the string*A1* and printed. Function *board-sq* returns a list containing the print name of a square index in algebraic chess notation. Note that in algebraic notation, the squares on a chess board are labeled as shown in Figure 12.1.

When executing the sample program, enter moves in algebraic notation. For example, to move the white kings's pawn forward two squares to start the game, enter the algebraic notation E2-E4.

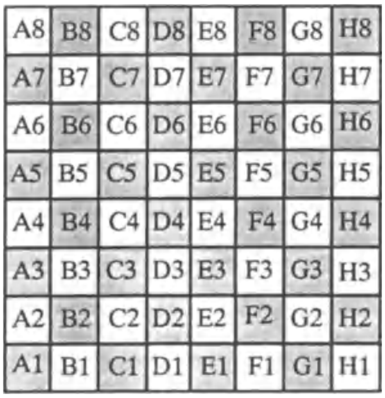


Figure 12.1 Algebraic Chess Notation used in Chess Program

The function *mover* is called by the main function *chess* to calculate and make a good move for the computer (black pieces). Function *mover* calls function *posib* to calculate all possible moves for the computer pieces and return these moves in a list. *Mover* then loops through all possible moves calling function *value* to calculate a static evaluation score. Note that internally, function *value* calculates all opponent responses so the equivalent of a two-ply lookahead is used for move selection. Function *posib* returns individual moves in one of three formats: a list containing "from" and "to" squares, the symbol *OO* (for king side castle), or the symbol *OOO* (for queen side castle). Much of the complexity in function *mover* is in handling castling. Notice that after *mover* evaluates a possible move, it restores the state of the board array **board** to its original state.

Function *chess* is the main driver for the chess program, looping to ask the opponent for a move and making a response computer move. As with function *mover*, most of the (limited) complexity in function *chess* lies in handling castling. Note that function *chess* takes an optional argument: if *chess* is called with a non-nil value, the game is continued without reinitializing the game state; this is useful for modifying and testing the program.

```
;;
; Chess Program by Mark Watson
;
; Bug list: (does not handle en passant capture)
;
; Copyright 1978-1990 by Mark Watson
;;

(proclaim
 '(special
   *wking-moved* *wrook1-moved* *wrook2-moved* *bking-moved*
   *brook1-moved* *brook2-moved* *move-num* *board*
```

```
  *human-square-control* *computer-square-control*
  *index* *piece* *value* *debug* *old-board* *old-moves*
  *moves-for-current-piece*))

;;
; Define global variables:
;;
(setq *board* (make-array (list 120))) ;; board
(setq *computer-square-control* (make-array (list 120)))
(setq *human-square-control* (make-array (list 120)))
(setq *moves-for-current-piece* nil)
(setq *index* '(0 12 15 10 1 6 0 0 0 6)) ;; piece index into move table
(setq *piece* '(0 -1 1 10 -10 0 1 -1 10 -10 -9 -11 9 11 0 8 -8 12 -12
                19 -19 21 -21 0 10 20 0 0 0 0 0)) ;; piece move table

;;
; Set value of the pieces: pawn, knight, bishop, rook, queen, and king:
;;
(setq *value* '(0 1 3 3 5 9 0 0 0 25))
;;
;  Turn on debug output:
;;
(setq *debug* t)

;;
; Remember last board state to allow replaying the last move
; when modifying the program:
;;
(setq *old-board* (make-array (list 120)) *old-moves* 0)

;;
;  Use constants to define piece values:
;;
(defconstant PAWN 1)
(defconstant KNIGHT 2)
(defconstant BISHOP 3)
(defconstant ROOK 4)
(defconstant QUEEN 5)
(defconstant KING 9)

;;
; Format of the board (e.g., square A1 is square 22 and
; square H8 is square 99 in the indexing scheme used
; for the board squares):
;;
```

```
; Square index:              Board with pieces:
; -------------------        -------------------------
;
;
; 92 93 94 95 96 97 98 99   BR  BN  BB  BQ  BK BB  BN  BR
; 82 83 84 85 86 87 88 89   BP  BP  BP  BP  BP  BP  BP  BP
; 72 73 74 75 76 77 78 79   .   X   .   X   .   X   .   X
; 62 63 64 65 66 67 68 69   X   .   X   .   X   .   X   .
; 52 53 54 55 56 57 58 59   .   X   .   X   .   X   .   X
; 42 43 45 45 46 47 48 49   X   .   X   .   X   .   X   .
; 32 33 34 35 36 37 38 39   WP  WP  WP  WP  WP  WP  WP  WP
; 22 23 24 25 26 27 28 29   WR  WN  WB  WQ  WK  WB  WN  WR

;; Note: Human (white) pieces are positive numbers and
;; computer (black pieces) are negative numbers. For
;; example, if (aref *board* 55) equals -2, then square
;; number 55 (D4 or Queen 4 in chess notation) has a
;; black knight. If it equals 2, then the piece on square
;; 55 is a white knight.
;;
; Initialize the board:
;;
(defun initChess ()
  (setq *wking-moved* nil *wrook1-moved* nil *wrook2-moved* nil)
  (setq *bking-moved* nil *brook1-moved* nil *brook2-moved* nil)
  (setq *move-num* 0)
  (dotimes (i 120)
    (setf (aref *board* i)
        (nth i '(
          7  7  7  7  7  7  7  7  7  7 ; empty squares around outside of board
          7  7  7  7  7  7  7  7  7  7 ; empty squares around outside of board
          4  2  3  5  9  3  2  4  7  7 ; white pieces
          1  1  1  1  1  1  1  1  7  7 ; white pawns
          0  0  0  0  0  0  0  0  7  7 ; 8 blank squares and 2 empty squares
          0  0  0  0  0  0  0  0  7  7 ; 8 blank squares and 2 empty squares
          0  0  0  0  0  0  0  0  7  7 ; 8 blank squares and 2 empty squares
          0  0  0  0  0  0  0  0  7  7 ; 8 blank squares and 2 empty squares
         -1 -1 -1 -1 -1 -1 -1 -1  7  7  ; black pawns
         -4 -2 -3 -5 -9 -3 -2 -4  7  7  ; black pieces
          7  7  7  7  7  7  7  7  7  7  7  7  7  7 ; empty squares
          7  7  7  7  7  7  7  7  7  7  7))))) ; empty squares
```

```
;;
; Print a piece. If the piece happens to be an empty square
; then print the representation for a black or white square on the board:
;;

(defun printPiece (piece blackSquare?)
 (if (zerop piece)
   (if blackSquare? (princ " X ") (princ " . "))
   (let ()
     (if (< piece 0)
       (princ " B")
       (princ " W"))
     (princ (nth (- (abs piece) 1)
             '("P" "N" "B" "R" "Q" "" "" "" "K"))))))

;;
; Print out the entire board:
;;

(defun printBoard ()
 (let ((startColumnList '(92 82 72 62 52 42 32 22)))
   (dotimes (i 8)
     (terpri)
     (dotimes (j 8)
       (let* ((boardPos (+ (nth i startColumnList) j))
             (blackSquare?
              (member boardPos
                     '(22 24 26 28 33 35 37 39 42 44 46 48
                       53 55 57 59 62 64 66 68 73 75 77 79
                       82 84 86 88 93 95 97 99))))
         (printPiece (aref *board* boardPos) blackSquare?))))
   (terpri)))

;;
; Calculate all possible moves from all squares:
;;

(defun posib (&aux returnedMoveList)
 (dotimes (ii 78)
   (let* ((i (+ ii 22))
         (boardVal (aref *board* i)))
     (if (< boardVal 0) ;; valid piece to move?
       ;; collect all squares to which piece on
       ;; square i can move to:
       (dolist (aMove (goto i t))
      (if (and
           ;; check for either an empty space or opponent piece:
           (>= (aref *board* (cadr aMove)) 0)
           (neq 7 (aref *board* (cadr aMove))))
         (setq returnedMoveList
            (cons aMove returnedMoveList)))))))
 (if (and
     (not *bking-moved*)
     (not *brook2-moved*)
     (equal (aref *board* 97) 0)
     (equal (aref *board* 98) 0)
     (< (aref *human-square-control* 96) 1)
     (< (aref *human-square-control* 97) 1)
     (< (aref *human-square-control* 98) 1))
   (setq returnedMoveList (cons 'oo returnedMoveList)))
 (if (and
     (not *bking-moved*)
     (not *brook1-moved*)
     (equal (aref *board* 95) 0)
     (equal (aref *board* 94) 0)
     (equal (aref *board* 93) 0)
     (< (aref *human-square-control* 96) 1)
     (< (aref *human-square-control* 95) 1)
     (< (aref *human-square-control* 94) 1))
   (setq returnedMoveList (cons 'ooo returnedMoveList)))
 returnedMoveList)

;;
; For a given square, return a list of all moves the
; piece on that square can move to:
;;

(defun goto (squareNum captureFlag
             &aux piece retList ival pieceType
             pieceIndex pieceMovementIndex)
 (setq piece (aref *board* squareNum)
     ival '(8 0 3)
     pieceType (abs piece)
     pieceIndex (nth pieceType *index*)
     pieceMovementIndex (nth pieceIndex *piece*))
 (if (not (equal piece 0)) ; make sure that there is a piece on squrae
   (case pieceType
    ((1)  ; PAWN
     (let ((sideIndex (if (< piece 0) -1 +1)))
      (dolist (captureDelta '(11 9)) ;; check for diagonal captures
       (let* ((movementOffsetInBoard (+ squareNum (* sideIndex captureDelta)))
              (targetPiece (aref *board* movementOffsetInBoard)))
         (if (or
              (and
               (<= targetPiece -1) ; enemy piece --> legal capture
               (neq targetPiece 7) ; not off of board
               (> piece 0))        ; computer piece moving
              (and
               (>= targetPiece 1)  ; computer piece
               (neq targetPiece 7) ; not off of board
               (< piece 0)))       ; player piece moving
           (setq retList (cons
                           (list
                             squareNum
                             (+ squareNum (* sideIndex captureDelta)))
                          retList)))))
     ;; Check for initial pawn move of two squares forward:
     (let* ((movementOffsetInBoard (+ squareNum (* sideIndex 20))))
       (if (and
            captureFlag
            (equal (aref *board* movementOffsetInBoard) 0) ; move-to sq empty?
            (equal (truncate (/ squareNum 10)) (if (< piece 0) 8 3))
            (if (< piece 0)
              (equal (aref *board* (- squareNum 10)) 0)
              (equal (aref *board* (+ squareNum 10)) 0)))
         (setq retList (cons (list squareNum
                             (+ squareNum (* sideIndex 20)))
                         retList))))
     (let* ((movementOffsetInBoard (+ squareNum (* sideIndex 10))))
       (if (and
            captureFlag
            (equal (aref *board* movementOffsetInBoard) 0)) ; move-to sq empty?
         (setq retList (cons (list squareNum
                             (+ squareNum (* sideIndex 10)))
                         retList))))))
   ((2 3 4 5 9)  ;;  KNIGHT BISHOP ROOK QUEEN KING
    (let* ((pieceType (abs piece))
           (movementTableIndex (nth pieceType *index*))
           (nextToSquare (+ squareNum (nth movementTableIndex *piece*))))
      (loop ;; over movement indices
        (loop
          (if (or
               (> nextToSquare 99)
               (< nextToSquare 22)
               (equal (aref *board* nextToSquare) 7))
              (return)) ;; break out of loop if OFF OF BOARD
            (setq retList (cons (list squareNum nextToSquare ) retList))
            (if (neq (aref *board* nextToSquare) 0) ; last move was a capture,
              (return)) so break out of the loop.
            (if (and
                (equal pieceType 1)
                (equal (truncate (/ squareNum 10)) 3))
              (return))
            (if (or
                (equal pieceType KNIGHT)
                (equal pieceType KING))
              (return))
            (setq nextToSquare
                  (+ nextToSquare
                     (nth movementTableIndex *piece*))))
          (setq movementTableIndex (+ movementTableIndex 1))
          ;; Lack of further  move segments is indicated
          ;; by a zero in the next element of the move
          ;; index table:
          (if (equal (nth movementTableIndex *piece*) 0)
            (return)) ;; no more move segments
          (setq nextToSquare
                (+ squareNum
                   (nth movementTableIndex *piece*)))))))
  retList)

;;
 ; Return the static evaluation value for a given board position:
 ;;

(defun value (toSq &aux retVal)
  (setq retVal 0)
  (dotimes (i 100)
    (setf (aref *computer-square-control* i) 0)
    (setf (aref *human-square-control* i) 0))
  ;; Calculate the number of times the computer's
  ;; pieces control each board square:
  (dotimes (ii 78)
    (let ((i (+ ii 22)))
      (if (< (aref *board* i) 0) ;; computer piece
        (let ((moveList (goto i nil))  pawnFudge)
          (dolist (move moveList)
            (if (equal (abs (aref *board* (car move))) 1)
              (setq pawnFudge 1.15)
              (setq pawnFudge 1))
          (setf (aref *computer-square-control* (cadr move))
                (+ (aref *computer-square-control* (cadr move)) pawnFudge)))))))
;; Calculate the number of times the player's
;; pieces control each square:
(dotimes (ii 78)
  (let ((i (+ ii 22)))
    (if (> (aref *board* i) 0) ;; computer piece
      (let ((moveList (goto i nil)) ;; generate moves from square # i
            pawnFudge)
        (dolist (move moveList)
          (if (equal (abs (aref *board* (car move))) 1)
            (setq pawnFudge 1.25)
            (setq pawnFudge 1))
          (setf (aref *human-square-control* (cadr move))
                (+ (aref *human-square-control* (cadr move)) pawnFudge)))))))
;; Subtract 1 from the control array
;; for the square being moved to:
(setf (aref *computer-square-control* toSq)
      (max 0 (- (aref *computer-square-control* toSq) 1)))
;; Set initial value based on board control:
(dotimes (ii 78)
  (let ((i (+ ii 22)))
    (setq retVal
          (+ retVal (* 0.1
                       (- (aref *computer-square-control* i)
                          (aref *human-square-control* i)))))))
;; Modify the value based on material advantage,
;; square control, and center control:
(dotimes (ii 78)
  (let ((i (+ ii 22)))
    (if (and
         (neq 7 (aref *board* i)) ;; not off of the board
         (neq 0 (aref *board* i))) ;; not a blank square
      (let ((control (- (aref *computer-square-control* i)
                        (aref *human-square-control* i))))
        ;; Piece value:
        (if (< (aref *board* i) 0)
          (setq retVal  ;; computer piece
                (+ retVal
                   (* (nth (abs (aref *board* i)) *value*)
                      8)))
          (setq retVal  ;; human piece
                (- retVal
                   (* (nth (abs (aref *board* i)) *value*)
                      8))))
        ;; Check for black piece on white-controlled square:
        (if (and (< control 0) (< (aref *board* i) 0))
          (setq retVal
               (- retVal
                 (* 14
                   (nth (abs (aref *board* i)) *value*)))))
        ;; Check for white piece on black-controlled square:
        (if (and (> control 0) (> (aref *board* i) 0))
          (setq retVal
               (+ retVal
                 (* 2
                   (min ; limit value of attacked piece to 3
                    3   ; points since this side to move next
                    (nth (abs (aref *board* i)) *value*))))))
        ;; King attacked:
        (if (and (> (aref *human-square-control* i) 0) (equal (aref *board* i) -9))
          (setq retVal (- retVal 5000)))
        ;; Queen attacked:
        (if (and (> (aref *human-square-control* i) 0) (equal (aref *board* i) -5))
          (setq retVal (- retVal 50)))))))
  ;; Pawn placement heuristics:
  (dolist (sq '(55 56 65 66)) ; loop over central four squares
    (if (equal (aref *board* sq) -1)
      (setq retVal (+ retVal 2))) ; black pawn
    (if (equal (aref *board* sq)  1)
      (setq retVal (- retVal 2)))) ; white pawn
  ; Loop over central 16 squares:
  (dolist (sq
        '(44 45 46 47 54 55 56 57 64 65 66 67 74 75 76 77))
    (if (equal (aref *board* sq) -1) (setq retVal (+ retVal 1))) ; black pawn
    (if (equal (aref *board* sq)  1) (setq retVal (- retVal 1)))) ; white pawn
  ;; Decrease value of moving queen in the first five moves:
  (if (and (< *move-num* 5) (equal (aref *board* toSq) -5))
    (setq retVal (- retVal 10)))
  ;; Decrease value of moving king in the first 15 moves:
  (if (and (< *move-num* 15) (equal (aref *board* toSq) -9))
    (setq retVal (- retVal 20)))
  (+ retVal (random 2)))

;;
; Convert internal square number to algrabraic notation:
;;

(defun board-pr (sq)
  (let* ((rank (truncate (/ sq 10)))
        (file (- sq (* rank 10))))
    (setq file (- file 2))
    (setq rank (- rank 1))
    (princ (nth file '("A" "B" "C" "D" "E" "F" "G" "H")))
    (princ rank)))

;;
; Return a list containing the algrabraic notation for a square.
;  For example, square number 22 would be converted to (A 1).
;;

(defun board-sq (sq)
  (let* ((rank (truncate (/ sq 10)))
        (file (- sq (* rank 10))))
    (setq file (- file 2))
    (setq rank (- rank 1))
    (list (nth file '("A" "B" "C" "D" "E" "F" "G" "H"))
          rank)))

;;
; Find the "best" move:
;;

(defun Mover ()
  (setq *move-num* (+ *move-num* 1))
  (let ((possibleMoves (posib))
        (bestMove nil)
        (bestValue -100000)
        (to nil)
        moveValues ;; for debug output only
        tosave fromsave newVal)
    (dolist (pm possibleMoves)
      (setq tosave 0)
      (if (equal pm 'oo)
        (let ()
          (setf (aref *board* 96) 0)
          (setf (aref *board* 97) -4)
          (setf (aref *board* 98) -9)
          (setf (aref *board* 99) 0)
          (setq to 10)) ; off of board
        (if (equal pm 'ooo)
          (let ()
            (setf (aref *board* 96) 0)
            (setf (aref *board* 95) -4)
            (setf (aref *board* 94) -9)
      (setq to 10) ; off of board
      (setf (aref *board* 92) 0))
    (let ()
      (setq fromsave (aref *board* (car pm)))
      (setq tosave (aref *board* (cadr pm)))
      (setf (aref *board* (cadr pm)) (aref *board* (car pm)))
      (setf (aref  *board*(car pm)) 0)
      (setq to (cadr pm)))))
;; Call value to calculate a numeric score for
;; how "good" this board position is for the computer:
(setq newVal (value to))
;; increase the score slightly for captures:
(if (> tosave 0)
  (setq newVal
       (+ newVal 12 (* 11 (nth tosave *value*)))))
(if (member pm '(oo ooo)) (setq newVal (+ newVal 10)))
(if *debug*
  (if (member pm '(oo ooo))
    (setq moveValues
         (cons
          (list (list pm) newVal)
          moveValues))
    (setq moveValues
         (cons
          (list
           (append
            (board-sq (car pm)) '(" to ")
            (board-sq (cadr pm)))
           newVal)
          moveValues))))
(if (> newVal bestValue)
  (let ()
    (setq bestValue newVal)
    (setq bestMove pm)))
(if (equal pm 'oo)
  (let ()
    (setf (aref *board* 96) -9)
    (setf (aref *board* 97) 0)
    (setf (aref *board* 98) 0)
    (setf (aref *board* 99) -4))
  (if (equal pm 'ooo)
    (let ()
      (setf (aref *board* 96) -9)
      (setf (aref *board* 95) 0)
      (setf (aref *board* 94) 0)
```

```
          (setf (aref *board* 92) -4))
        (let ()
          (setf (aref *board* (car pm)) fromsave)
          (setf (aref *board* (cadr pm)) tosave)))))
  (if *debug*
    (let ()
      (dolist (x
              (sort moveValues
                    #'(lambda (x y) (> (cadr x) (cadr y)))))
        (terpri)
        (dolist (y (car x))
          (princ y))
        (princ " : ")
        (princ (truncate (cadr x))))
      (terpri)))
  (if (< bestValue -1000)
    'checkmate
    bestMove)))

;;
; Main driver program:
;;
(defun chess (&optional restart)
  (if (not restart) (initChess))
  (if (equal restart 'backup)  ;; debug option to back up one move on restart
    (let ()
      (princ "Backing up the game by one move")
      (terpri)
      (dotimes (i 120)
        (setf (aref *board* i) (aref *old-board* i)))
      (setq *moves-for-current-piece* *old-moves*)))
  (printBoard)
  (loop
    (princ "Enter your move (e.g., d2-d4) : ")
    (let* ((response (princ-to-string (read)))
           (moved? nil))
      (if (equal response "OO") ;; Common LISP converts reads to uppercase
        (if
          (and  ;; let us be sure that casting king side is legal
           (not *wking-moved*)
           (not *wrook2-moved*)
           (equal (aref *board* 27) 0)
           (equal (aref *board* 28) 0)
           (< (aref *computer-square-control* 26) 1)
           (< (aref *computer-square-control* 27) 1)
```

```
       (< (aref *computer-square-control* 27) 1))
      (let ()
        (setq moved? t)
        (princ "Castle King side") (terpri)
        (setf (aref *board* 26) 0) ; blank old king square
        (setf (aref *board* 29) 0) ; blank old rook square
        (setf (aref *board* 27) 4) ; rook
        (setf (aref *board* 28) 9))) ; king
    (if (equal response "OOO") ;; Common LISP converts reads to uppercase
      (if
        (and ;; be sure that a queen side castle is legal
         (not *wking-moved*)
         (not *wrook1-moved*)
         (equal (aref *board* 23) 0)
         (equal (aref *board* 24) 0)
         (equal (aref *board* 25) 0)
         (< (aref *computer-square-control* 26) 1)
         (< (aref *computer-square-control* 25) 1)
         (< (aref *computer-square-control* 24) 1))
        (let ()
          (setq moved? t)
          (princ "Castle Queen side") (terpri)
          (setf (aref *board* 26) 0) ; blank old king square
          (setf (aref *board* 22) 0) ; blank old rook square
          (setf (aref *board* 25) 4) ; rook
          (setf (aref *board* 24) 9))))) ; king
  (if (equal (length response) 5)
    (let*
      ((fromCol (- (char-code (char response 0)) 63))
        (fromRow (- (char-code (char response 1)) 47))
        (toCol (- (char-code (char response 3)) 63))
        (toRow (- (char-code (char response 4)) 47))
        (from (+ (* fromRow 10) fromCol))
        (to (+ (* toRow 10) toCol)))
      (setq moved? t)
      (setf (aref *board* to) (aref *board* from))
      (setf (aref *board* from) 0)))
  (if (not moved?)
    (print "What???")
    (let ()
      (printBoard)
      ;; Remember last state of the board to allow backing up
      ;; one move for debug by running: (chess 'backup):
      (dotimes (i 120)
        (setf (aref *old-board* i) (aref *board* i)))
```

```
(setq *old-moves* *moves-for-current-piece*)
;; Calculate the "best" computer move:
(let ((bestMove (Mover)))
  (if (equal bestMove 'checkmate)
    (let ()
      (princ "Checkmate!!")
      (terpri)
      (return)))
  (if (equal bestMove 'oo)
    (let ()
      (setf (aref *board* 96) 0)
      (setf (aref *board* 97) -4)
      (setf (aref *board* 98) -9)
      (setf (aref *board* 99) 0)
      (setf *bking-moved* t)
      (setf *brook2-moved* t)
      (terpri) (princ "OO") (terpri))
    (if (equal bestMove 'ooo)
      (let ()
        (setf (aref *board* 96) 0)
        (setf (aref *board* 95) -4)
        (setf (aref *board* 94) -9)
        (setf (aref *board* 92) 0)
        (setf *bking-moved* t)
        (setf *brook1-moved* t)
        (terpri) (princ "OOO") (terpri))
      (let ()
        (setf (aref *board* (cadr bestMove)) (aref *board* (car bestMove)))
        (setf (aref *board* (car bestMove)) 0)
        (terpri)  (princ "Computer move : ")
        (board-pr (car bestMove))
        (princ "-")
        (board-pr (cadr bestMove))
        (terpri)))))
 (printBoard))))))
```

12.4 Executing the Chess Program

Since the heuristics that are used to control move selection are sound, most of the moves that this program makes are good. However, the program easily succumbs to tactical traps such as knight forks (that is, positioning a knight so that it simultaneously attacks two pieces of greater value). The following output from a LISP listener window shows the beginning of a game with the variable ***debug*** set to a true value:

```
? (setq *debug* t)
T
? (chess)

 BR BN  BB BQ BK BB BN  BR
 BP  BP  BP BP  BP  BP BP  BP
  .   X   .   X   .   X   .   X
 X  .   X  .   X  .   X   .
  .   X   .   X   .   X   .   X
 X  .   X  .   X  .   X   .
 WP WP WP WP WP WP WP WP
 WR WN WB WQ WK WB WN WR
Enter your move (e.g., d2-d4) : e2-e4

 BR BN  BB BQ BK BB BN BR
 BP  BP  BP BP  BP  BP BP BP
  .   X   .   X   .   X   .  X
 X  .   X  .   X  .   X  .
  .   X   .   X  WP X   .  X
 X  .   X  .   X  .   X  .
 WP WP WP WP  . WP  WP WP
 WR WN WB WQ WK WB WN WR

E7 to E5 : 0
G8 to F6 : -1
C7 to C5 : -1
C7 to C6 : -1
E7 to E6 : -2
B8 to C6 : -2
D7 to D6 : -2
B7 to B6 : -2
G7 to G6 : -2
G7 to G5 : -2
A7 to A6 : -2
F7 to F6 : -2
G8 to H6 : -3
A7 to A5 : -3
H7 to H6 : -3
D7 to D5 : -11
F7 to F5 : -14
B7 to B5 : -16
H7 to H5 : -16
B8 to A6 : -45

Computer move : E7-E5
```

```
BR BN BB BQ BK BB BN BR
BP BP BP BP X  BP BP BP
.  X  .  X  .  X  .  X
X  .  X  .  BP .  X  .
.  X  .  X  WP X  .  X
X  .  X  .  X  .  X  .
WP WP WP WP .  WP WP WP
WR WN WB WQ WK WB WN WR

Enter your move (e.g., d2-d4) :
```

In this output, the value of each possible move is printed in sorted order, with best moves listed first. The following output from a LISP listener window shows the beginning of a game with the variable **debug** set to a false value:

```
? (setq *debug* nil)
NIL
? (chess)

BR BN BB BQ BK BB BN BR
BP BP BP BP BP BP BP BP
.  X  .  X  .  X  .  X
X  .  X  .  X  .  X  .
.  X  .  X  .  X  .  X
X  .  X  .  X  .  X  .
WP WP WP WP WP WP WP WP
WR WN WB WQ WK WB WN WR
Enter your move (e.g., d2-d4) : c2-c4

BR BN BB BQ BK BB BN BR
BP BP BP BP BP BP BP BP
.  X  .  X  .  X  .  X
X  .  X  .  X  .  X  .
.  X  WP X  .  X  .  X
X  .  X  .  X  .  X  .
WP WP .  WP WP WP WP WP
WR WN WB WQ WK WB WN WR

Computer move : E7-E5
```

```
BR BN BB BQ BK BB BN BR
BP BP BP BP X  BP BP BP
.  X  .  X  .  X  .  X
X  .  X  .  BP .  X  .
.  X  WP X  .  X  .  X
X  .  X  .  X  .  X  .
WP WP .  WP WP WP WP WP
WR WN WB WQ WK WB WN WR
Enter your move (e.g., d2-d4) : b1-c3

BR BN BB BQ BK BB BN BR
BP BP BP BP X  BP BP BP
.  X  .  X  .  X  .  X
X  .  X  .  BP .  X  .
.  X  WP X  .  X  .  X
X  .  WN .  X  .  X  .
WP WP .  WP WP WP WP WP
WR .  WB WQ WK WB WN WR

Computer move : F8-B4

BR BN BB BQ BK X  BN BR
BP BP BP BP X  BP BP BP
.  X  .  X  .  X  .  X
X  .  X  .  BP .  X  .
.  BB WP X  .  X  .  X
X  .  WN .  X  .  X  .
WP WP .  WP WP WP WP WP
WR .  WB WQ WK WB WN WR
Enter your move (e.g., d2-d4) :
```

To castle king side, enter *OO* when asked for a move, to castle queen side, enter *OOO* . The chess-playing program has little input error checking so enter your moves carefully. Note that you can stop the program at any time, modify it, and restart it in the current game position by executing *(chess T)*.

12.5 Projects

1. The example program only plays the black pieces. Modify the program to ask the human player which side to play. The easiest way to make this modification is to change the printing functions to reverse the board, while keeping the same internal representation for the squares and pieces.

2. This program does not allow en passant pawn captures. Modify the program to play a legal game of chess by removing this deficiency adding en passant move generation logic to function *posib*.

3. If you have a good LISP compiler and a fast computer, you may want to try expanding the limited lookahead capability of this program.

   The following project is not too difficult and will prevent the program from falling into most simple tactical traps:

   Add a Shannon type B heuristic search to this program. Only add one ply to the search by using the following approach: the example program sorts the moves in descending order of merit when the debug option is turned on. Use this logic to capture the best *N* moves for the computer. For each of these *N* moves, make the move and switch white and black pieces in the array **board**, then evaluate the "best" move for the opponent. For each of the best original *N* moves being evaluated, set the merit score for the move to its original value minus the merit score for the best opponent response.

4. Modify the debug printout in the sample program to print out the numeric rating for each possible computer move under evaluation and also the factors that contribute to the rating. For example, calculate the rating *value* in the static evaluation function as

   ```
   (setq value
         (+ materialValue mobilityValue centerControlValue
            kingControlValue))
   ```

   and print out each value term.

5. Add heuristics to detect pawn weaknesses (e.g., doubled pawns, backward pawns, and pawns around both kings when they are not on their original squares) to the evaluation function.

6. Modify the sample program to play against itself. Use the same functions for both sides except for the move evaluation function *value*. When you alter the move heuristics, you can play the new set of heuristics against the previous set of heuristics.

13 Introduction to Chaos Theory

This chapter provides a short overview of Chaos theory. We will see how even simple systems that are modeled with **nonlinear** equations, that is, those containing polynomial or exponential terms, can show surprising behavior. A simple population growth model will demonstrate a chaotic system.

13.1 Modeling the Physical World

The advancement of science includes not only a refinement of our understanding of the physical world, but also the improved understanding of what it is that we do not know. Newtonian physicists viewed the universe as if it were a complex clock: if we study it hard enough, master the physical laws of nature, and can measure the initial conditions of the universe with sufficient accuracy then total understanding and predictability are possible. The Heisenburg uncertainty principle and Einstein's theory of relativity shattered this hope and desire for the predictable nature of the universe by forcing us to accept that the universe is not measurable with sufficient precision and that instantaneous travel to conduct remote measurements is impossible. With the benefit of hindsight, we now see that science has often concentrated on problems that can be accurately represented by linear systems of equations (and thus solved) and stayed away from difficult (and often unsolvable) nonlinear problems.

Chaos theory is a new science (Gleick, 1987) which attempts to illuminate the nonlinear processes involved in predicting natural phenomena such as weather and population growth. Nonlinear systems are characterized by extreme sensitivity to initial conditions; making a very small change in one of the initial conditions of a nonlinear system often results in radically different behavior. For example, a small change in the atmosphere in one part of the world can have major and unpredictable effects elsewhere. There is hope of understanding nonlinear systems. In dealing with nonlinear systems, the emerging science of Chaos theory gives us a framework for understanding what we can and can not successfully model.

13.2 Example Population Model Programs

As a simple example of a chaotic (nonlinear) system, biologist Robert May modeled fish population (Gleick, 1987) with the simple equation:

$$Population_{N+1} = lambda * Population_N * (1 - Population_N)$$

The parameter *lambda* controls the rate of population growth. The truly amazing thing about this simple equation is its chaotic behavior as *lambda* is slowly increased in value.

The first sample program produced the diagram in Figure 13.1. (The graphics library from chapter 2 must be loaded to run this program.) The main loop in the first example program iterates the value of the y-coordinate in the graphics window (*y-axis*).

The population growth rate *lambda* increases in value with the y-coordinate *y-axis*. For every y-plotting coordinate, the first example program iterates setting the population value *population* using the above equation for population growth. The x-plotting coordinate is calculated for each iteration of the above population growth equation by scaling the population, multiplying it by 500.02.

```
;;
; File: May-model.lisp
;
; Population growth model of Robert May: generates a bifurcation diagram
; showing large scale effect of the population growth rate.
; Note: the area to the upper right of the initial curve indicates
; extinction due to the rate of population growth being too small
; (the variable lambda).
;
;  Copyright 1990 by Mark Watson
;;

(defun bifur ()
  (init-plot "Global Population Growth Bifurcation Diagram" 502 230)
  (clear-plot)
  (show-plot)
  (plot-string 236 22 "Extinction")
  (plot-string 16 41 "Steady state")
  (plot-string 334 123 "Period doubled")
  (plot-string 190 215 "Chaos")
  (let ((x-axis)
       (lambda) (x 0.1) (population))
    (dotimes (y-axis 198)
      (setq lambda (* 4 (+ 0.20 (/ y-axis 250.0))))
      (dotimes (iter 198)
        (setq population (* lambda x (- 1 x)))
        (setq x-axis (truncate (* population 500.02)))
        (if (and  (> x-axis 0)  (< x-axis 501))
          (plot-line x-axis y-axis x-axis y-axis))
        (setq x population)))))
```

The example of population growth shows the extreme sensitivity of nonlinear systems to initial conditions. In Figure 13.1, the top of the diagram illustrates the area of small growth rate where population increases gradually. As the growth rate

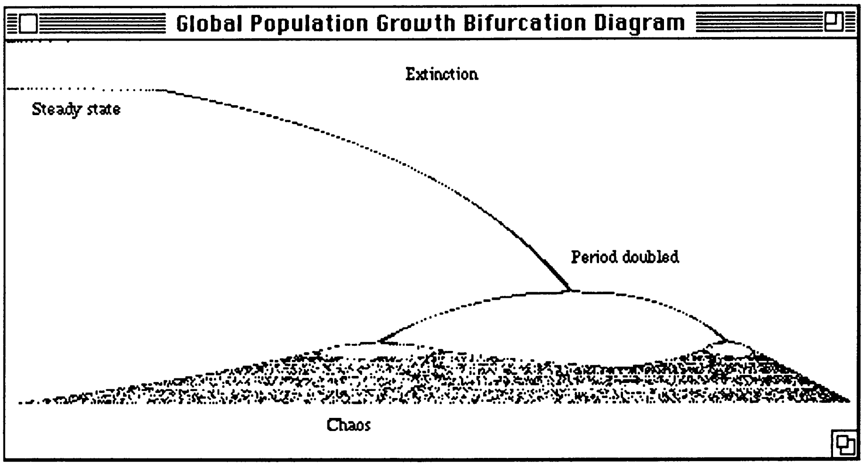


Figure13.1. Effect of population growth rate on population. Vertical axis shows *lambda* increasing from 0 (top of plot) to the value of 1 (at the bottom of the plot). Horizontal axis represents the simulated population (larger values to the right side of the plot).

lambda increases, frequency doubling occurs: the population tends to oscillate between two values. As the growth rate continues to increase (lower in the picture), frequency doubling occurs again: the population oscillates between four values. As *lambda* continues to increase, the population becomes chaotic and unpredictable.

The following program listing generated the diagram in Figure 13.2 which magnifies the region around the area where the period of population growth oscillation doubles, and the area where the population growth becomes chaotic. The graphics library from chapter 2 must be loaded before running this program. This second example program is nearly identical to the first. The plot title is different and the growth rate calculation for *lambda* is different.

```
;;
; File: May-model-closeup.lisp
;
; Generates a magnified bifurcation diagram magnifying the area
; where period doubling turns into chaos.
;
; Copyright 1990 by Mark Watson
;;

(defun bifur ()
 (init-plot "Magnification of Bifurcation around Period Doubling/Chaotic area"
          502 237) ;; specify title and x,y size of plot window
 (clear-plot)
 (show-plot)
 (let ((x-axis)
     (lambda) (x 0.1) (population))
```

```
  (dotimes (y-axis 220)
    (setq lambda (* 4 (+ 0.85 (/ y-axis 1500.0))))    ;;; change the growth rate
    (dotimes (iter 198)
      (setq population (* lambda x (- 1 x)))
      (setq x-axis (truncate (* population 500.02)))
      (if (and  (> x-axis 0)   (< x-axis 501))
        (plot-line x-axis y-axis x-axis y-axis))
      (setq x population)))))
```

In Figure 13.2, notice the clear, nonchaotic areas near the bottom of the diagram. These are regimes in which the system becomes stable and oscillates between three possible values. If the sample program which created Figure 13.2 is modified to magnify this stable regime, the resulting plot will look surprisingly like the original diagram in Figure 13.1. Indeed, this repeatability is infinitely deep as we keep magnifying areas of the plot where there is a stable regime. This self-similarity will be seen again in the Fractal images produced by the example programs in Chapter 14.

The example of population growth shows both how quickly a nonlinear system becomes unpredictable and how brute force numerical techniques will sometime fail. Experiments with the constants embedded in the sample programs (that is, the 198 iterations on the population equation and the expression setting the value for *lambda*) like those suggested below will convince the reader just how radically the output of nonlinear models varies with the input parameters!

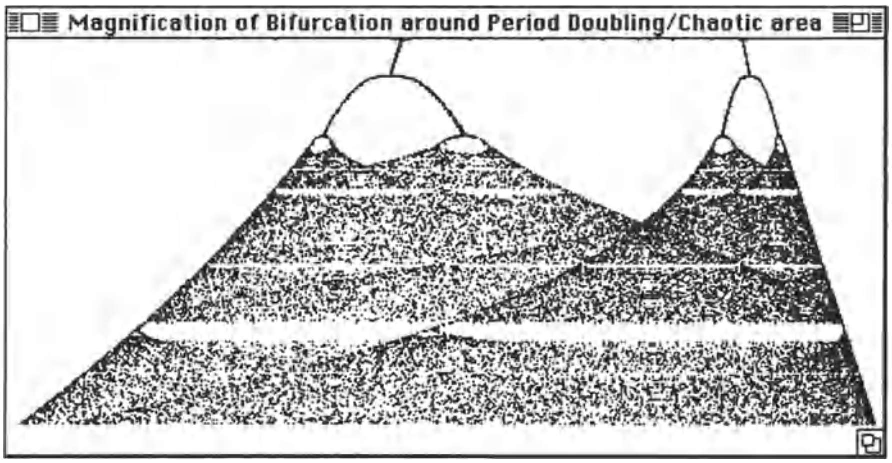


Figure 13.2. Effect of population growth rate on population. Magnification of the area showing frequency doubling (top of the picture) and chaotic behavior (bottom of picture).

13.3 Projects

1. Modify the program which produced Figure 13.2 to magnify the stable regime near the bottom of the diagram. You should see three structures each looking similar to the diagram in Figure 13.1. Repeat this process a couple of times to convince yourself that the frequency doubling/chaotic structure is infinitely repeatable and self-similar.

2. Delta rule neural networks display chaotic behavior in organizing their internal states. Use the complete neural network simulator listed at the end of chapter 3 to perform the following experiment:

   a. Create a small neural network using: *(NewDeltaNetwork '(2 3 2))*.
   b. Make a copy of this network. Make small changes to a few of the initial random weights in the copied network.
   c. Train both neural networks using the same training data. How do the weights in the two networks compare? How does the pattern recognition performance of the two networks compare?

14
Fractal Images

We are accustomed to thinking of the dimensionality of space as being an integer number. Distance along a line is measured in one dimension, as position on a piece of paper is measured in two dimensions, and a point in space is measured in three dimensions. For IBM mathematician Benoit Mandlebrot, space can have fractional dimension in the study of **fractals**. Fractal plots can be seen in generated images in motion pictures and they find practical applications in the modeling of blood vessels and turbulence. Blood vessels form a fractal space. Mapped out, they fit in a three-dimensional space but they take up relatively little volume in our bodies. Like the population growth plots seen in chapter 13, the pattern of blood vessels seems infinitely repeatable as large arteries branch into increasingly smaller capillaries. However, there is a lower limit to the size of blood vessels while a mathematical model of a fractals repeats forever as we magnify its fractal space.

Mathematicians Michael Barnsley and Alan Sloan use fractal **iterated function systems** (IFS) to compact digitized images. Their company (appropriately named IFS Systems) is developing technology using IFS encoding to obtain data compression ratios much larger than traditional compression schemes. Data compression techniques will become increasingly important when the digital **ISDN** (integrated services digital networks) communications standard is implemented and all information (television, radio, information services) received in our homes and offices will be digital. Effective methods for data compression will help to make personalized television and information retrieval both possible and affordable. Image reconstruction given compact IFS codes is computationally inexpensive; however, converting images to IFS codes (Barnsley, 1988) is computationally expensive.

Two types of fractal plots will be presented in this chapter: Mandelbrot set plots and IFS plots. As the two example programs illustrate, both types of plots are **self-similar**, that is, like the images produced by the population growth model in Chapter 13, image features repeat with surprising regularity as they are magnified.

14.1 Mandelbrot Set

The fractal geometries that Benoit Mandlebrot has made a popular pastime for so many personal computer owners solve a very simple iterative function. This function is simply:

$$Z_{time+1} = Z_{time} * Z_{time} + C$$

Z is a complex variable and *C* is a complex constant. Complex numbers contain two numbers. a *real* and an *imaginary* component. The easiest way to understand a complex number is to think of it as representing a single point on an x-y plot; by convention the x-coordinate of the point is the *real* value, and the y-coordinate of the point is the *imaginary* value.

Consider a point *C* on the complex plane. In the above equation, set *Z* equal to *C*. The next value for the complex variable is calculated by squaring *Z* and then adding the point *C*. If we can iterate in this way *forever* without the magnitude of *Z* exploding to infinity, the original point *C* on the complex plane is a member of the Mandlebrot set. To plot this set of points, we will show every point *C* in the region around the complex number zero as black if the point is in this set; for points *C* that are not in the Mandelbrot set we will count the number of iterations required to reach a large threshold value and use this number of iterations to select a gray-scale or color value to plot the point C on the complex plane.

The graphics library listed in chapter 2 provides the capability to plot in five gray-scale values. The variable *MAX_COLORS* should be set to 5 for compatibility with the library listed in chapter 2. If your computer can produce a larger number of discrete colors or gray-scale values, modify the function *plot-fill-rect* in the graphics library to support the maximum number of gray-scale values your system supports and modify the value of *MAX_COLORS*. For example, if your computer supports 16 colors, modify function *plot-fill-rect* to take a range of colors between 0 and 15 and set the value of *MAX_COLORS* in the first sample program in this chapter to 16.

The first sample program, which plots the Mandelbrot set, has two modifiable parameters: *num-x-cells* and *num-y-cells*. Increasing these parameters increases the number of grid cells in the horizontal and vertical directions in the plots. The parameter *cell-width* sets the number of pixels to use to represent each complex number that we test for membership in the Mandelbrot set.

Function *M* is the main function for calculating and plotting the Mandelbrot set. It loops over a range of points on the complex plane and iterates a maximum of 60 times for each point to test for set membership.

A single run of the following program generated the Mandelbrot set plots in Figures 14.1 and 14.2.

```
;;
 ; File: Mandelbrot.lisp
 ;
 ; Calculate and plot the Mandelbrot set
 ; Copyright 1990 by Mark Watson
 ;;


;;
 ; Define the maximum number of gray-scale or color values
 ; that function plot-fill-rect uses as a pattern (or color) range:
 ; (Plot-fill-rect is defined in the graphics library in Chapter 2.)
 ;;
(setq MAX_COLORS 5)
```

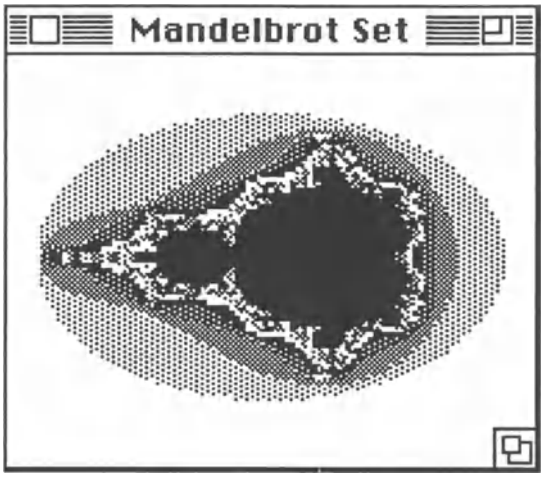


Figure 14.1. Low-resolution Mandelbrot Set Plot. This was produced by setting *num-x-cells* to 90 and *num-y-cells*.

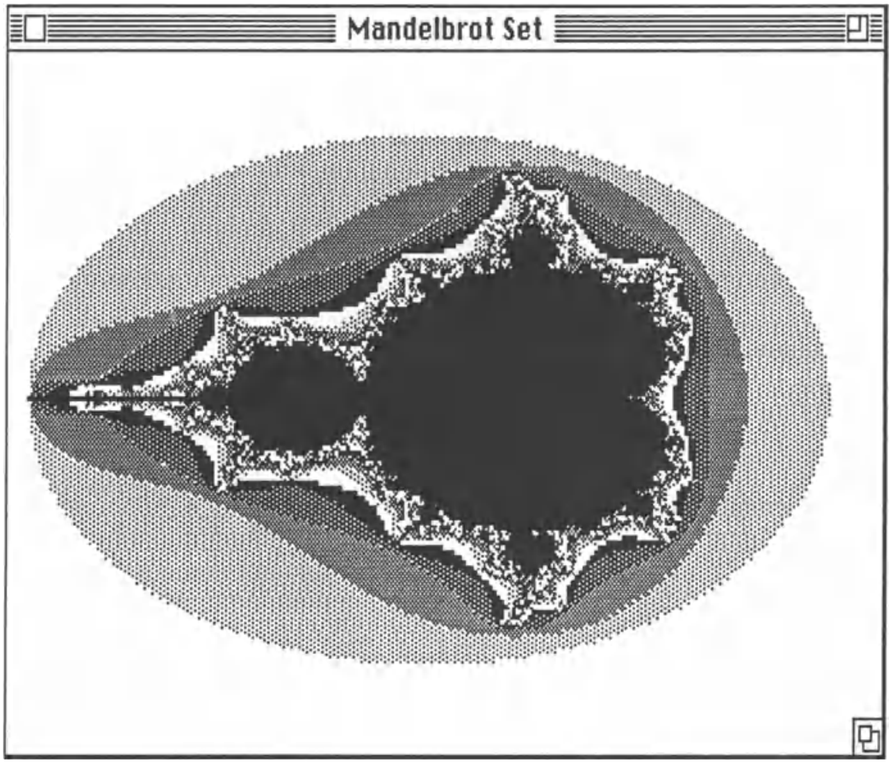


Figure 14.2. Medium-resolution Mandelbrot Set Plot. This was produced by setting *num-x-cells* to 220 and *num-y-cells* to 130.

```
;;
 ; Define the number of complex points to evaluate for Mandelbrot
 ; set membership: a rectangle of points num-x-cells by num-y-cells:
 ;;

(setq num-x-cells 90     num-y-cells 65)

;;
 ; Set the cell width and height for plotting a single point
 ; on the complex plane:
 ;;

(setq cell-width 2)
(setq delta-x-cell (/ 3.2 num-x-cells)    delta-y-cell (/ 3.2 num-y-cells))

;;
 ;  Function M  is called to calculate the Mandelbrot set
 ; for complex points around the complex number zero.
 ;;

(defun M (&optional (debug nil))
  (init-plot "Mandelbrot Plot"
           (+ 10 (* cell-width num-x-cells))
           (+ 25 (* cell-width num-y-cells)))
  (clear-plot)
  (show-plot)
  (let ((x 0.5) (y 0.5) (z (complex 0 0)))
    (dotimes (ix num-x-cells)
      (setq x (- (* ix delta-x-cell) 2.0))
      (dotimes (iy num-y-cells)
        (setq y (- (* iy delta-y-cell) 1.6))
        (setq z (complex x y))
        (let ((index nil))
          (dotimes (i 60)
            (setq z (+ (complex x y) (* z z)))
            (setq index i)
            (if (>  (+  (* (realpart z) (realpart z))  (* (imagpart z) (imagpart z))) 4)
              (setq i 1000)))
          (plot-fill-rect
           (+ 10 (* ix cell-width))
           (+ 10 (* iy cell-width))
           cell-width cell-width (mod index MAX_COLORS)))))))
```

14.2 Iterated Function Systems

Iterated function systems (IFS) can be used to greatly compress digital images. In a sense, generating IFS plots is similar to generating Mandelbrot set plots. Random points are selected on a complex plane and moved by iterating a simple set of functions which determine the path of this point on the complex plane. The randomly chosen points are allowed to move around the complex plane for several cycles, then each point on the path is plotted (in the program listed in this section) as a black dot on the screen. Michael Barnsley's book on fractals (Barnsley, 1988) provides many examples of iterated function systems.

The following program uses **affine** (Barnsley, 1988) transformations in two-dimensions to recreate complex images from a simple set of coefficients. In one dimension, an affine transformation is of the form: $f(x) = ax + b$, where a and b are constants. The one-dimensional affine transformation f transforms an interval $[0, L]$ to an interval $[b, b+a*L]$ if $a>0$ or the interval $[b-a*L, b]$ if $a<0$. As an example, if $a=-1$ and $b=2$, then the interval [0,1] is mapped to [1,2] with the order of the points in the interval reversed.

A two-dimensional affine transformation (Barnsley, 1988) is used in the following program and is of the form: $g(x,y) = (ax + by + e, cx + dy + f)$, where a, b, c, d, e, and f are constants. This transformation will convert any point on a two-dimensional plane to a new point. The following program uses four distinct affine transformations to draw an image. At each iteration though the main plotting loop, one of the four affine transformations is randomly selected.

The following program uses IFS codes (sets of the coefficients a, b, c, d, e, and f) for producing two simple geometric images: a sunflower and a fern. Data is defined at the top of the program for these two images. The variable *LIMIT* defines the number of iterations used to restore the images. Setting *LIMIT* to a small value generates a dim outline of the image. As *LIMIT* is increased, the image is filled in. The function *IFS* uses the graphics library from chapter 2 to plot the restored IFS images. Function *IFS* magnifies a small portion of the restored image to illustrate the self-similarity of IFS images. After copying the data for either the sunflower or fern, the main program loop draws *LIMIT* points in the graphics window. To draw each point, a random number between 0 and 1 is generated and compared to the entries in *probability-table* to determine which of four affine transforms to use for the current iteration. The IFS coefficients a, b, c, d, e, and f are used to calculate x and y plot coordinates *newx* and *newy*. The first 20 times through the main loop, the x and y plot coordinates are updated with no plotting performed. After the first 20 iterations in the main loop, the x-y plot coordinates are scaled to the size of the graphics window and a single point is plotted.

```
; ; File: IFS.lisp
 ;
 ; Iterated function system for recreating images from IFS tables
 ;
 ; Copyright 1990 by Mark Watson
 ;;

;;
; Data specification for two IFS designs: a Sunflower
; and a Fern. Set one flag (only) from the following
; list to a non-nil value to produce either the Sunflower
; or Fern design:
;;

(setq SUNFLOWER nil)
(setq FERN T)

(if SUNFLOWER
  (setq title "IFS Example - Sunflower with Magnification"
      XSCALE 40
      YSCALE 26
      XORG 260
      YORG 180
      maxTable 4
      a '(-0.30  -0.30    0.02  0.00)
      b '(-0.60  -1.00    0.01  0.02)
      c '(1.10    1.0     0.00  0.00)
      d '(0.11    0.010   -0.2  0.00)
      e '(4.00    4.00    0.00  0.01)
      f '(0.00    0.00    0.00  0.02)
      p '(0.02    0.95    0.01  0.02)
      x1 1.2 y1 0.8 x2 1.8 y2 2.1 ;; define zoom box
      XZOOMORG -270 YZOOMORG 340
      zx1 (+ (* x1 XSCALE 6) XZOOMORG)
      zy1 (- YZOOMORG (* y1 YSCALE 6))
      zx2 (+ (* x2 XSCALE 6) XZOOMORG)
      zy2 (- YZOOMORG (* y2 YSCALE 6))))

(if FERN
  (setq  title "IFS Example - Fern with Magnification of Leaf"
       XSCALE 75
       YSCALE 21
       XORG 250
       YORG 260
       maxTable 4
```

```
        a '(0.85 -0.15 0.2 0.0)
        b '(0.04 0.28 -0.26 0.0)
        c '(-0.04 0.26 0.23 0.0)
        d '(0.85 0.24 0.22 0.16)
        e '(0.0 0.0 0.0 0.0)
        f '(1.6 0.44 1.6 0.0)
        p '(0.85 0.065 0.065 0.02)
       x1 -0.85 y1 6.1 x2 -0.6 y2 7.1 ;; define zoom box
       XZOOMORG 390 YZOOMORG 900
       zx1 (+ (* x1 XSCALE 6) XZOOMORG)
       zy1 (- YZOOMORG (* y1 YSCALE 6))
       zx2 (+ (* x2 XSCALE 6) XZOOMORG)
       zy2 (- YZOOMORG (* y2 YSCALE 6))))

(setq LIMIT 150000)  ;; limit on iterations

;;
; IFS program:
;;
(defun IFS ()
  (let ((pt 0.0)
       (probability-table '(0 0 0 0 0 0))
       x y pk newx newy)

    ;; Ppen a graphics window:
    (init-plot title 470 270)

    ;; Plot a small "capture" box to enlarge in the zoom box:
    (plot-line
     (+ (* x1 XSCALE) XORG) (- YORG (* y1 YSCALE))
     (+ (* x2 XSCALE) XORG) (- YORG (* y1 YSCALE)))
    (plot-line
     (+ (* x2 XSCALE) XORG) (- YORG (* y1 YSCALE))
     (+ (* x2 XSCALE) XORG) (- YORG (* y2 YSCALE)))
    (plot-line
     (+ (* x2 XSCALE) XORG) (- YORG (* y2 YSCALE))
     (+ (* x1 XSCALE) XORG) (- YORG (* y2 YSCALE)))
    (plot-line
     (+ (* x1 XSCALE) XORG) (- YORG (* y2 YSCALE))
     (+ (* x1 XSCALE) XORG) (- YORG (* y1 YSCALE)))

    ;; Plot the outline of the enlarged zoom box:
    (plot-line zx1 zy1 zx2 zy1)
    (plot-line zx2 zy1 zx2 zy2)
    (plot-line zx2 zy2 zx1 zy2)
```

```
(plot-line zx1 zy2 zx1 zy1)

;; Scale the probabilities:
(dotimes (i maxTable)
  (setq pt (+ pt (nth i p)))
  (setf (nth i probability-table) pt))

;; Set the initial position of a point:
(setq x 0.0 y 0.0)

;; Main iteration loop:
(dotimes (i LIMIT)
  (setq pk (random 1.01))
  (if (<= pk (nth 0 probability-table))
    (setq k 0)
    (if (<= pk (nth 1 probability-table))
      (setq k 1)
      (if (<= pk (nth 2 probability-table))
        (setq k 2)
        (setq k 3))))
  (setq newx (+
            (* (nth k a) x)
            (* (nth k b) y)
            (nth k e)))
  (setq newy (+
            (* (nth k c) x)
            (* (nth k d) y)
            (nth k f)))
  (setq x newx  y newy)

  ;; Throw way the first 20 points to wait for stabilization:
  (if (> i 19)
    (let ((ix (truncate (+ (* x XSCALE) XORG)))
          (iy (truncate (- YORG (* y YSCALE)))))
      ;; Plot only the first 30000 points for the entire
      ;; design to prevent over-saturation of points.
      ;; Note that we plot more points in the
      ;; enlarged zoom box since the points are
      ;; spread out on the screen by a factor of 6.
      ;; in each direction.
      (if (< i 30000)
        (plot-line ix iy ix iy))
      ;; Check to see if point (ix,iy) is inside the zoom box:
      (if (and
           (> x x1)
        (< x x2)
        (> y y1)
        (< y y2))
    (let ((ixx (truncate (+ (* x XSCALE 6) XZOOMORG)))
          (iyy (truncate (- YZOOMORG (* y YSCALE 6)))))
      (plot-line ixx iyy ixx iyy))))))))
```

The data for two designs is provided at the top of the example program listing; changing the values of the LISP variables *FERN* to *nil* and *SUNFLOWER* to *T* in this example program creates the sunflower design shown in Figure 14.3.

The screen dump in Figure 14.4 shows a picture of a fern produced by reconstruction using four IFS codes (see the variables *a*, *b*, *c*, *d*, *e*, *f*, and *p* in the program listed above). Note that in the upper left corner of both plots displays a magnification of part of the larger image. Both IFS generated plots are self-similar: any small segment of the plot is made up of features that resemble the larger segment.

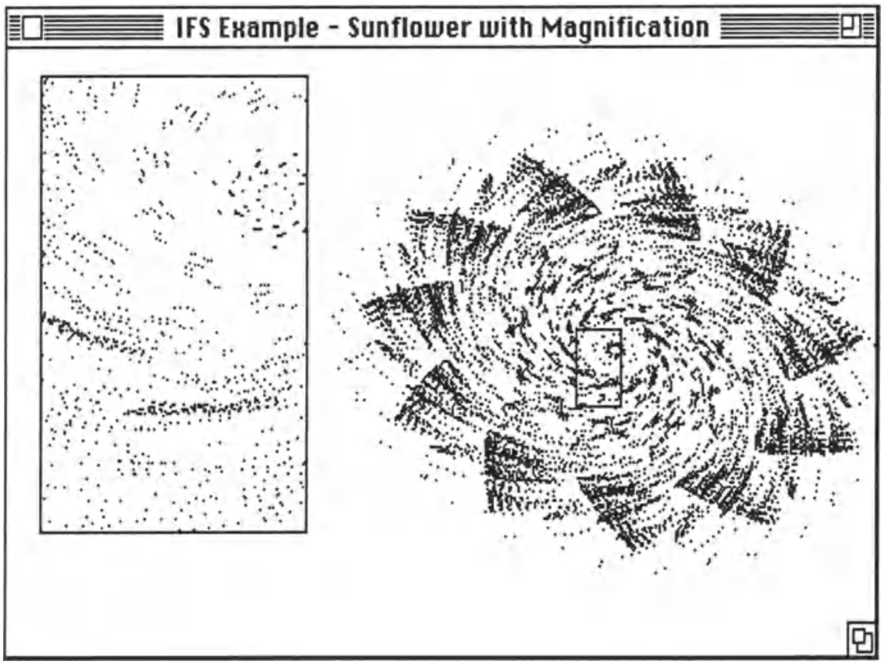


Figure 14.3. A Sunflower Produced by an IFS.

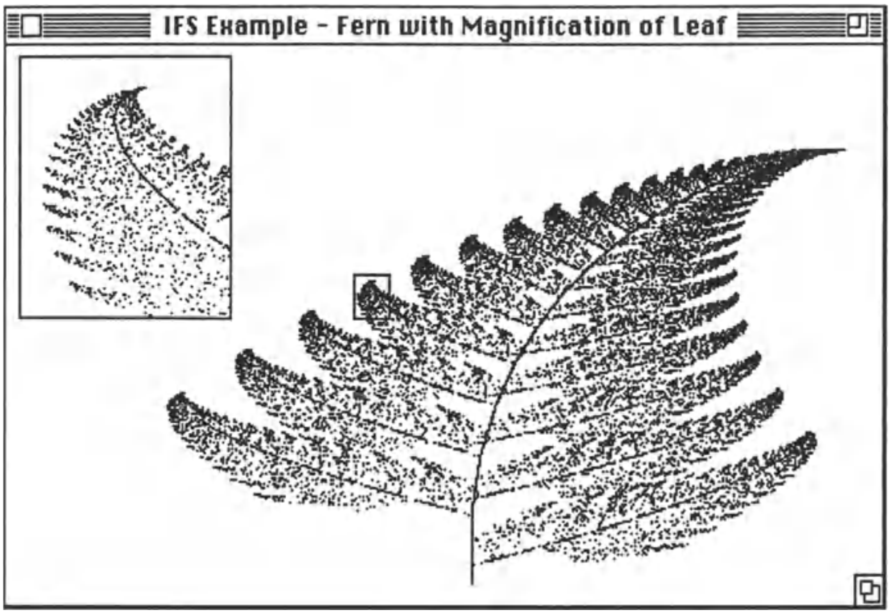


Figure 14.4. A Fern Produced by an IFS.

14.3 Projects

1. The first example program plots the entire Mandelbrot set. By changing the initial values for x and y, and the *delta-x* and *delta-y* values in the program, you can enlarge any portion of the complex plane. Choose a complex point (as specified by an x and y coordinate pair) of interest in the Mandelbrot set plot and modify the example program to magnify the Mandelbrot set plot in an area centered at your selected point and with any specified size.

2. Experiment with the constants a, b, c, d, e, and f used in the IFS program for generating the sunflower and fern images. If you make small changes to the constants you will notice that the IFS is fairly stable, that is, in general the images will not change drastically in appearance as the constants are slowly changed.

Annotated Bibliography

Barnsley, Michael: *Fractals Everywhere.* San Diego (CA): Academic Press, Inc. 1988. This book provides a formal treatment of fractals and dynamic systems.

Carpenter, Gail A. and Grossberg, Stephen: "ART 2: Self-Organization of Stable Category Recognition Codes for Analog Input Patterns." *Applied Optics*, 1987. Formal presentation of adaptive resonance theory with neural model defined by differential equations describing network dynamics.

Frey, Peter W.: *Chess Skill in Man and Machine.* New York (NY): Springer-Verlag, Inc. 1977. Reference for computer chess. Covers theory, implementation, and example games played by computer chess programs.

Gleick, James: *Chaos, Making a New Science.* Harrisonburg (VA): Penguin Books Ltd. 1987. Describes the people responsible for the development of chaos theory.

Hart, T., Nilsson, N., Raphael, B.: "A Formal Basis for the Heuristic Determination of Minimum Cost Paths." IEEE Transactions on Systems Science and Cybernetics, 1968. Description of the A* search algorithm.

Kohonen, Teuvo: *Self-Organization and Associative Memory.* Beltz Offsetdruck (Germany): Springer-Verlag, Inc. 1984. Provides an analysis of basic associative memory devices and associative memory.

Rumelhart, David E., and McClelland, James L. (editors): *Parallel Distributed Processing*, Volume 1. Cambridge (MA): MIT Press 1986. Provides background material for neural network models, activation functions, and steepest-descent learning procedures.

Schank, Roger C. and Riesbeck, Christopher K.: *Inside Computer Understanding.* Hilsdale (NJ): Lawrence Erlbaum Associates, Inc. 1981. This book is built around a collection of miniature Lisp programs which demonstrate key concepts of conceptual dependency theory: semantic-based parsing, scripts, and natural language generation.

Shannon, Claude E.: "Programming a Computer for Playing Chess" 1950: Philosophical Magazine, volume 41.

Waibel, A.; Hanazawa, T.; Hinton, G.; Shikano, K.; and Lang, K.: "Phoneme Recognition Using Time-Delay Neural Networks." Tokyo: ATR Laboratory report, 1987. This technical report gives a summary of the results and implementation details of time-delay neural networks.

Winston, Patrick Henry: *Artificial Intelligence.* USA: Addison-Wesley 1977. Classic introduction to artificial intelligence and Lisp programming.

Winston, Patrick Henry and Horn, Berthold Klaus: *Lisp.* USA: Addison-Wesley 1981. Another classic introduction to the Lisp programming language with many useful small examples.

Woods, W. A.: "Procedural Semantics for a Question-Answering Machine." Proceedings of AFIP Volume 33, 457-471 1968. Description of using ATN parsers in a natural language question-answering system.

Index